Open
Microprocessor systems
Initiative

Arches

Application, Refinement and Consolidation of HIC Exploiting Standards

ESPRIT Project 20693

Deliverable D.3.1.1

Report on Interfacing Commodity processors to IEEE1355 DS Links

CERN

Classification: Public
ISSUE DATE: March 1998

D 3.1.1 Report on interfacing Commodity processors to IEEE1355 DS links

Executive Summary.

Description of task

Task 3.1 was defined in terms of porting the successful DS link technology away from the defunct T9000 processor it had been designed for and towards commodity processors for which a high performance interconnect market still existed.

This deliverable, which is in three parts covers:

- the study of baseline criteria and performance issues.
- the design and implementation of the hardware platform.
- the design and implementation of the software and firmware.

Effort

Within the Arches workpackage this task was estimated at one man-year of effort and yet the deliverable clearly represents work well in excess of this. This extra work has been accounted for outside this project.

Dissemination and technology transfer

The results of the study and development have also been published at conferences such as RT97:

http://www.cern.ch/HSI/dshs/marcel/marcel.ps

and WOTUG-21:

http://www.cern.ch/HSI/dshs/publications/wotug21/dsnic/main.html

as well as being publicised on our public WEB pages:

http://www.cern.ch/HSI/dshs/

Summary

While the DS links have not had the success hoped for by their industrial sponsors they continue to enjoy some limited success as a standardised component in the European Space Agency program. As a result of the successful work achieved in this task, and its public exposure, several contractors for the ESA program have approached CERN with requests for technology transfer. These requests are currently under negotiation.
We consider this a good reflection on the quality of the work achieved in this task.

D 3.1.1 Report on interfacing commodity processors to IEEE 1355 DS links

PART 1: The study of baseline interfacing architectures and their performance issues

1 Introduction

The DS links which form part of the IEEE 1355 standard[1] were originally integrated onto the T9000 Transputer chip[2] to provide interprocessor communications. The integration of the processor and its memory with four DS links was extremely efficient by virtue of a communications processor, the Virtual Channel Processor (VCP), which implemented packet and message passing protocols, as well as multiplexing multiple virtual links onto physical links. The overall communications performance was aided by the very low process context switching time of the T9000. The result being very low message passing overheads and good usage of the available link bandwidth.

However, despite the T9000's impressive communications capability, by the time it was in production it was unable to compete computationally with state of the art microprocessors. The T9000 has been used successfully as an I/O processor to provide DS link capability to other microprocessors with significantly higher computational performance. An example presented within this report is the DEC Alpha, but with the cessation of production and supply of the T9000 announced by SGS Thomson it has become important to find a replacement strategy for interfacing commodity processors to DS links.

The migration towards other processors had to be carefully considered. We had to analyse the communications performance of the T9000 in order to identify the strengths and weaknesses of the architecture, both as stand-alone and as a communications co-processor to the DEC-Alpha. A simple interface using a host processor to drive a DS link directly has also been implemented, i.e. the VCP functionality is provided by the host. The experience gained from the investigation and development of these systems allows the key design issues of a network interface to be identified. All of these issues were fed into the design and development of an efficient DSNIC (DS Link Network Interface Controller) which is presented in the second and third parts of this deliverable.

This report first presents as a baseline the performance of the T9000 driving DS links. Factors contributing to low message latency and high data throughput are identified. The use of the T9000 as an I/O processor for a DEC Alpha microprocessor is then discussed. Following this the performance of a PowerPC emulating the VCP functionality of the T9000 in software is presented. Finally the important design issues of a network interface (applicable to any serial link technology) are identified.

2 T9000 Communication Performance

This section presents an evaluation of the communications performance of the T9000 and the factors affecting performance are identified. The performance of the T9000 is the baseline by which other DS interfacing implementations can be judged. Further details on the communications performance of the T9000 can be found in [3,4].

2.1 Single link performance

The message passing overhead (elapsed time to send a zero length message) between two directly connected 20 MHz T9000s was measured to be 7.5 µsecs. If the T9000s are connected via a STC104 switch [5] then the overhead is 9.6 µsecs.

In Figure 1 the dependency of the bandwidth between two 20 MHz T9000s on the number of virtual links used is presented. The T9000s are connected via a single STC104 switch. The figure shows the bandwidth as a function of message size for one to five virtual links mapped onto a single physical link. The bandwidth represents the usable amount of data exchanged between two T9000s running at 20 MHz. The discontinuity in bandwidth at each 32 byte boundary is due to the packetisation performed by the VCP. Data is transferred in packets of maximum length 32 bytes and each individual packet on a virtual link must be acknowledged before another is transmitted. A 32 byte message requires a single packet to be sent and acknowledged, whereas a 33 byte message requires two packets to be sent and acknowledged.

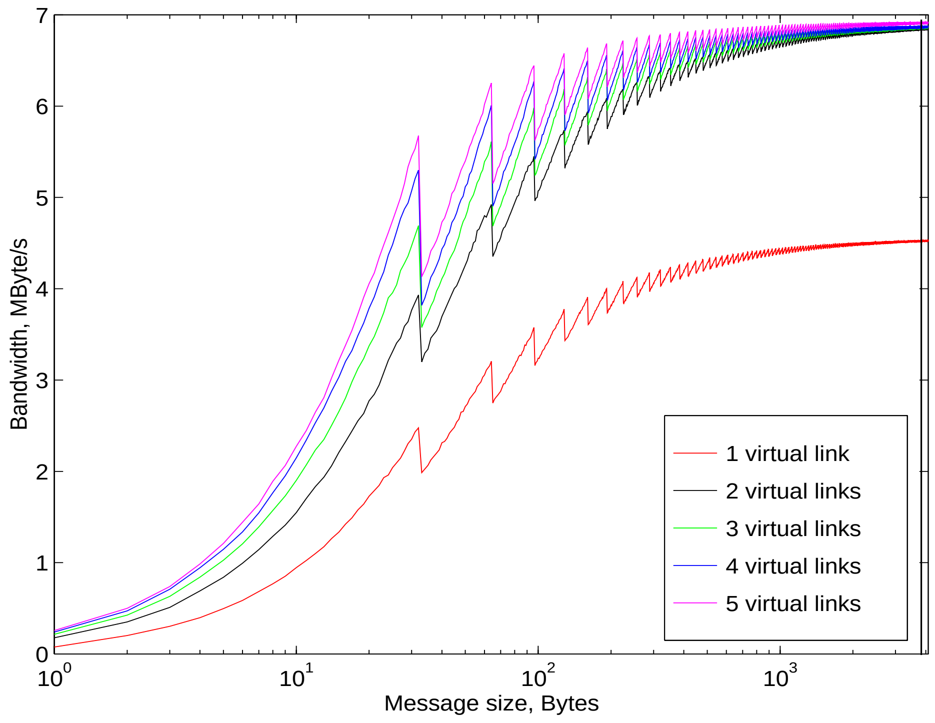

FIGURE 1 **Single link bandwidth, uni-directional. 20 MHz T9000s, 100 Mbits/s links, 64 bit memory interface, 8K cache 8K internal memory, connected via single STC104.**

In Figure 1 the increase in bandwidth for more virtual links can be accounted for by the increased packet inter-leaving performed by the VCP and more efficient use of its pipelined architecture. The VCP inputs the data on multiple virtual links from the link and writes it directly to the required area of user memory, it can then re-schedule the relevant process requiring that data. When multiple virtual links are used, packets for different virtual links may be transmitted independently of the reception of acknowledge packets on other virtual links. If a single virtual link is in use, a packet is sent, then the virtual link (and physical link) is idle until the acknowledge is received, which reduces the bandwidth. If another virtual link is in use, the first virtual link (waiting for an acknowledge) stays idle but the second can use the link. Directly connected T9000s achieve a higher bandwidth over a single virtual link (6.4 Mbytes/s) because the acknowledges pass between the T9000s in less time. A constant message start up time accounts for the reduced bandwidths for smaller messages.

The theoretical maximum of a DS link running at 100 Mbits/s using 2 byte headers is 9.26 Mbytes/s [6]. The measured limit is only 7.0 Mbytes/s. This is due to the fact that the VCP is only running at 20 MHz, whereas it was designed to achieve full link throughput at above 30 MHz.

The elapsed time for sending a message is composed of two parts: a fixed message passing overhead (T_0) and the time to transfer the message, which is the message length (L) divided by the effective transfer rate ($R_{effective}$). The relation can be expressed as

$$ElapsedTime = T_0 + \frac{L}{Reffective}$$

The message passing overhead is defined as the elapsed time to transfer a zero length message from an application program in the source node to an application program in the destination node. For the T9000 implementation this value is 7.5 microseconds.

The message passing overhead can be divided into two further components: the network latency and the node message latency. The network latency is defined as the time to propagate a single byte through the switching network. The node message latency is defined as the I/O initiation time in a source or destination.

The equation allows the calculation of two very important parameters for quantifying the performance of parallel computers: the message passing overhead and the achieved asymptotic bandwidth. For ideal performance the $R_{effective}$ in the above equation should be the raw transmission speed of the interconnection medium. In Figure 2 the elapsed time to send a message on a single virtual link is plotted against message length. In addition, the ideal performance is shown where $R_{effective}$ is the raw transmission speed of the link. The raw transmission rate of a 100 Mbits/s link is approximately 10 Mbytes/s (one byte sent as a 10 bit token over the link).

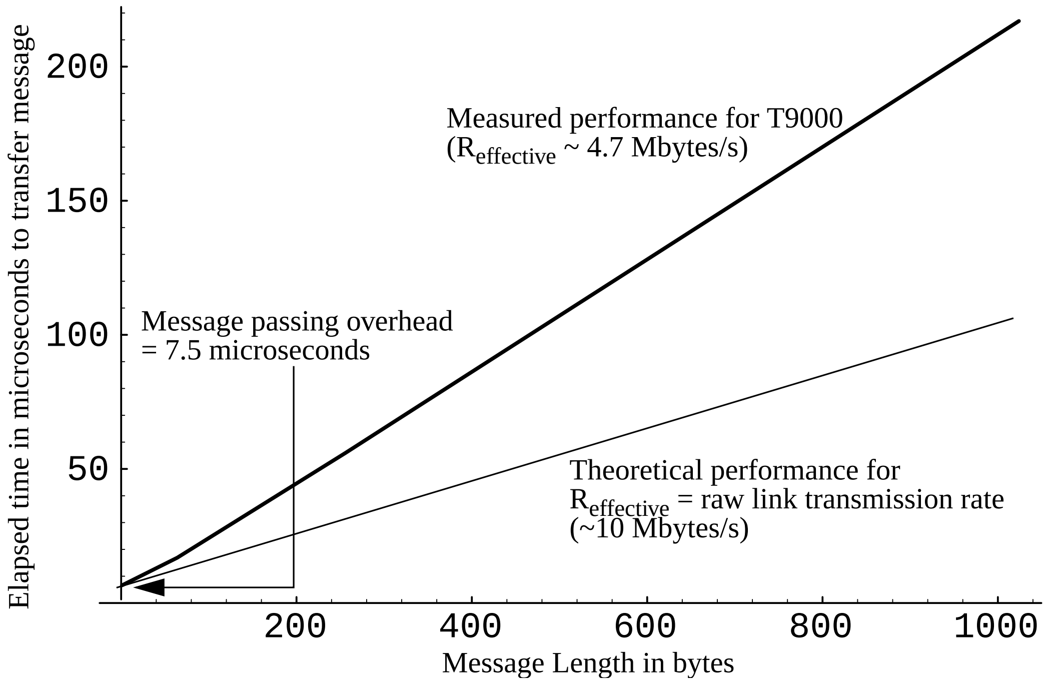

FIGURE 2 **Measured and theoretical results for the T9000 (20 MHz T9000s using single virtual link connected via a single STC104)**

It is clear that the achieved $R_{effective}$ is not equal to the raw transmission rate of the link. Reasons for this are (applicable to any link interface):

- Node interface to the raw link, i.e. the rate at which data can be transferred into the application program from the link - the low clock speed of the VCP on the T9000 (20 MHz) limits performance to 7 Mbytes/s (for multiple virtual links)
- Protocol implementation overheads, e.g. packetisation of messages and production of acknowledges - this limits the performance to 4.7 Mbytes/s for a single virtual link, due to the strict acknowledge scheme of the T9000

2.2 Dependence on memory bandwidth

If data is written to the links from internal memory of a 20 MHz T9000 the limit when four links are in use is 28 Mbytes/s. Again, the bottleneck is the VCP which can only drive the links at 28 Mbytes/s. If data is accessed from external memory the limit is 16.5 Mbytes/s. This is due to a combination of the poor design of the external memory interface and the low clock speed (20 MHz compared to 50 MHz design).

2.3 One to four physical links

Figure 3 demonstrates that the maximum achieved bandwidth scales linearly with the number of physical links used for uni-directional traffic, i.e. the rate at which the VCP can drive the links scales linearly with the number of links in use.

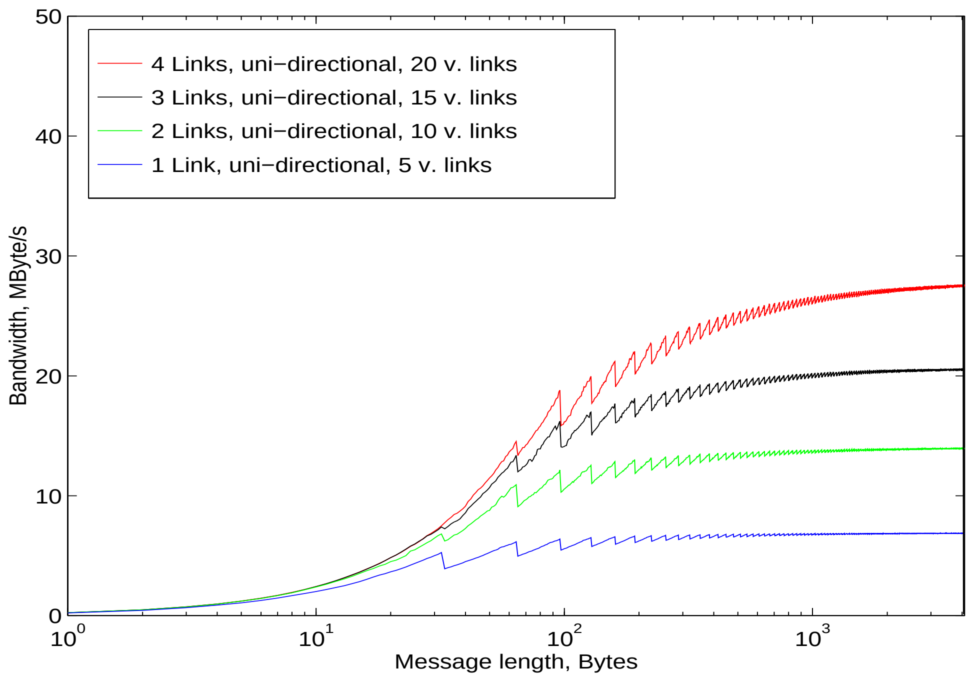

FIGURE 3 **Multiple links, 20 MHz T9000s, all data in internal memory, 100 Mbits/s links. Connections via a single STC104.**

2.4 Dependence on clock speed

The measured bandwidths as a function of message size when using 20 and 25 MHz processors are shown in Figure 3 (20 MHz) and Figure 4 (25 MHz). The theoretical limit for the single link bandwidth is 9.26 Mbytes/s on each physical link. The bandwidths measured at 20 and 25 MHz fall short of the 9.26 Mbytes/s, but there is a clear improvement from the 20 MHz to 25 MHz processors. This demonstrates the improvement of increasing the speed of the VCP, however, it is still unable to exploit fully the capacity of the links. However, if the T9000 were running at 30 MHz the VCP should be able to reach the theoretical limits for the link bandwidth.

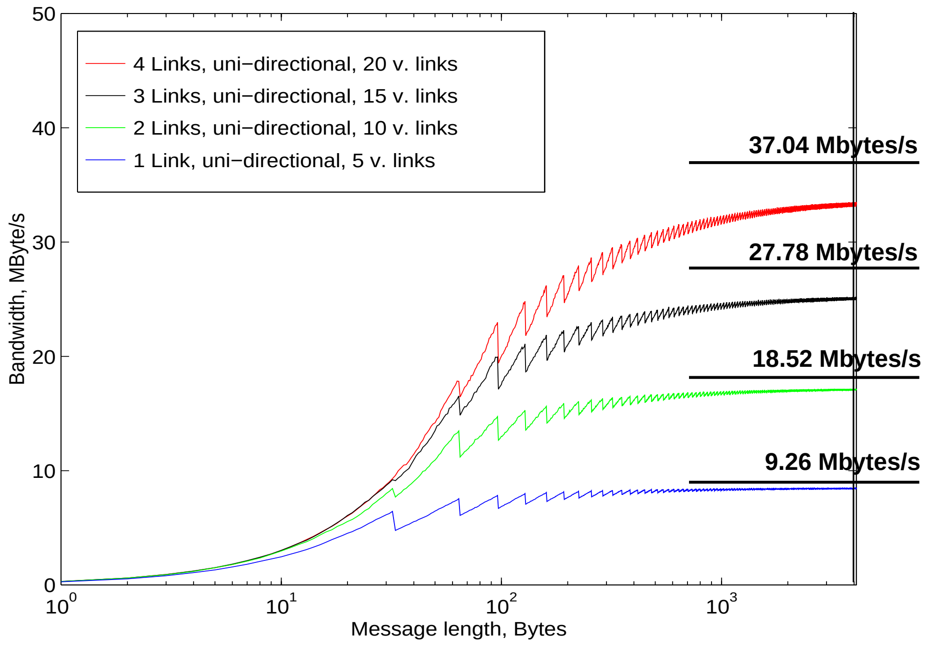

FIGURE 4 **Multiple links, 25 MHz T9000s, all data from internal memory, 100 Mbits/s links. Connections via STC104s.The theoretical limits of 1 to 4 DS links are also shown.**

2.5 Interrupt Response and Context Switch Summary

The T9000 contains a hardware scheduler providing efficient interrupt response and context switch times. The T9000 interrupt response time (time from interrupt until user specified code is executing) is 1.9 µs. This is the time for a full process context switch, i.e. a context switch which may occur when any instruction is being executed. A partial context switch (or timeslice operation) which may only occur at certain instructions requires 1.4 µs.

2.6 Short Message Sends

An investigation has been carried out to measure the number of short messages that a single T9000 can produce per second. Ten processes run concurrently on a single source and a single destination, each process on the source is communicating to a single process on the destination via a virtual link. The source processor sends in parallel on all virtual links, continually switching between the processes, sending as many short messages as possible.

On the destination the processes continually wait for input. When a packet arrives at the destination the corresponding process for that virtual link will be rescheduled.

The performance depends heavily on the efficiency of the VCP and context switching times. Results are produced from two 20 MHz T9000s with a single STC104 switch between them. The following parameters are varied: message length, number of virtual links and number of physical links. This investigation is of interest for two reasons:

- It evaluates the ability to context switch during communications. When short messages are used the latencies of context switching are more dominant than when larger messages are in use. Ten processes are continuously context switching on a single processor every time they send a single packet (message).
- It measures the capability of the processor to perform as a centralised supervisor or control processor in a distributed system, where many short messages must be sent to large numbers of possible destinations.

Figure 5 shows the performance when using up to four physical links. The maximum rate at which messages can be sent is 430 KHz, which occurs for 0 to 32 byte messages.

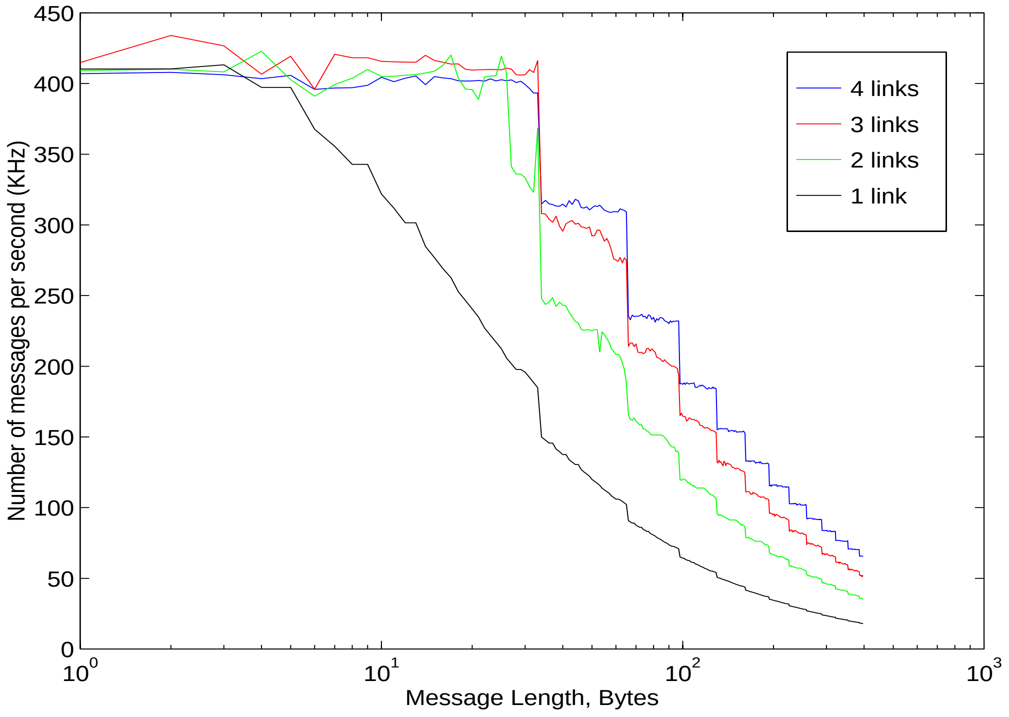

FIGURE 5 **Short message sends, vary number of physical links, between two 20 MHz T9000s connected by STC104s, 64 bit interface 8 K cache 8 K internal memory. 100 Mbits/s links. 10 virtual links in use per physical link.**

At 430 KHz the total CPU time required to produce each message is 2.33 µs. This corresponds to 1.4 µs for a partial context switch and less than 1 µs of CPU time to perform the communication. This performance is possible due to extremely low context switch times and the ability of the VCP to multiplex multiple virtual links onto the physical links without loading the CPU.

2.7 Combined Communication and Computation

An important aspect of performance is the load which communications place on the CPU. There are two important times to consider for a communication: the actual time of the communication and the time the CPU is utilised.

The following test has been carried out: a single processor runs two processes, one performs floating point operations and the other performs communication over a single virtual link to a second processor. The processors are connected via a single STC104. The number of floating point instructions carried out by the first process is measured during communication, this gives the fraction of the CPU which is available. The message length is varied (which alters the communications bandwidth) and the load of this communication on the CPU is measured. Measuring the load communication imposes on the CPU is the same as measuring the extent to which communications and computation can be overlapped. The results are shown in Figure 6. It is clear that the load on the CPU is below 100% for all message lengths, for 10 Kbyte messages the load on the CPU is ~20%. The reduction in CPU load as the message length increases is due to a constant message passing overhead on the T9000. The main factors which allow the T9000 to achieve such results are:

- The VCP (a dedicated on-chip communications processor) which off-loads protocol and general communications requirements from the CPU. The VCP also facilitates a low message passing overhead and efficient use of the available link bandwidth (it avoids memory copies)
- Low context switch times (provided by a hardware scheduler) to allow the computation process to take full advantage of the remaining CPU time.

The increase in CPU load above 10 Kbytes is due to a cache effect, the T9000 uses 8 Kbytes of cache for the measurements.

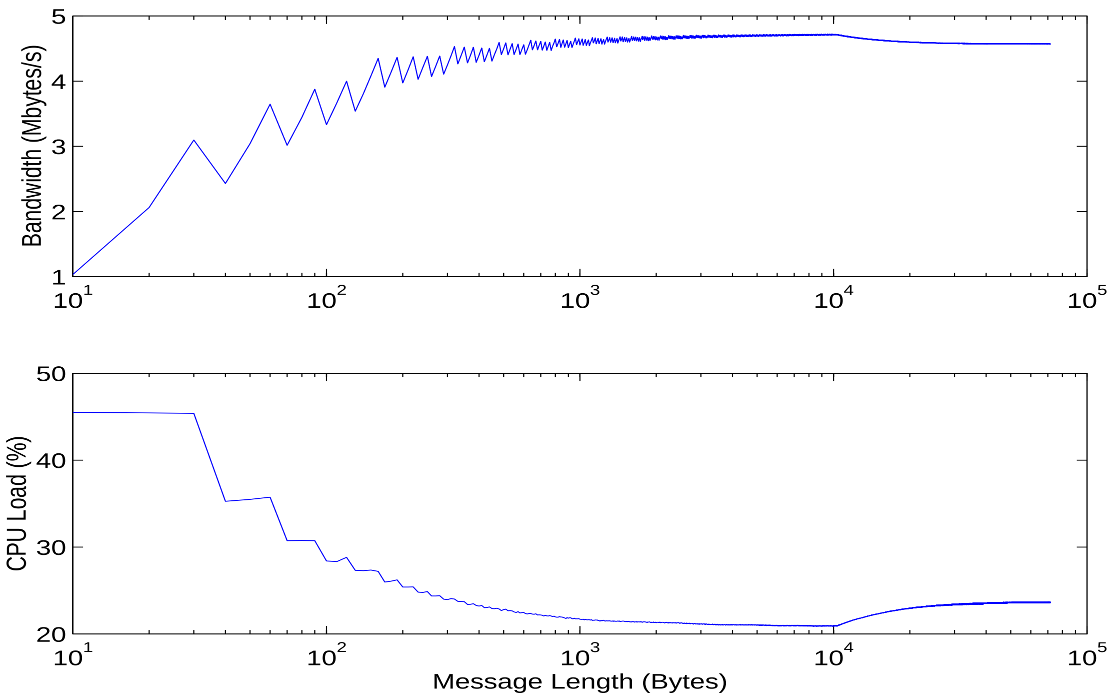

FIGURE 6 Comparison of throughput and communications Load on CPU.

2.8 Summary

The T9000 communications results show extremely low message passing overheads (i.e. 7.5 µs between two directly connected processors). The 25 MHz T9000 is very close to exploiting the full bandwidth of 4 DS links, a 30 MHz T9000 could saturate the links fully. The T9000 can produce over 400,000 short messages per second and for longer messages the communications load on the CPU is approximately 20%. Primarily, this performance has been possible due to low context switching times and the dedicated communications processor (the VCP).

3 The T9000 used as a communications co-processor

The TransAlpha module[7] was designed to remedy the lack of computing power provided by the T9000. The module consists of a T9000 Transputer, acting now solely as a communications controller, and a DEC 21066A Alpha processor used to boost computational performance.

In order to facilitate the inclusion of this module into existing networks the module has a mother-card housing the T9000 which conforms to the INMOS HTRAM specification. A daughter-card with the DEC Alpha and its memory plugs onto the mother-card via a PCI bridge.

A block diagram of the TransAlpha module is shown in Figure 7.

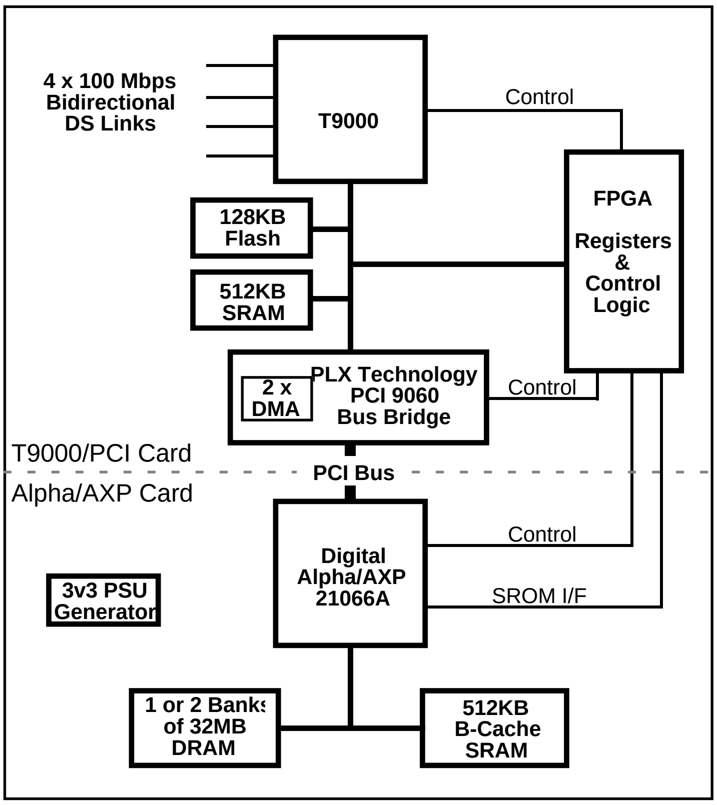

FIGURE 7 **A block diagram of the TransAlpha module showing the main components and the Mother-card - Daughter-card split.**

The major components are:

- A 233MHz 21066A DEC Alpha with 512 KBytes of second level cache and 32 or 64 MBytes of DRAM. This chip has an on-chip PCI IO controller connected directly to a PCI bus. A serialROM interface provides a path for the initialisation data.
- A 20 MHz T9000 Transputer with 512 KBytes of SRAM on its local bus.
- A PLX PCI 9060 chip acting as a bridge between the PCI bus and the local bus of the T9000.
- An FPGA providing glue logic between the local bus of the T9000 and the PCI 9060 chip. This is also responsible for initialising the Alpha chip.

Both processors have full access to all memory via either their own local bus or the PCI bridge. Access is either via single word read and writes or via DMA transfers. DMA transfers are performed by the PCI 9060 chip which has two independent DMA engines. These are controllable from either processor. Interrupts may be sent to either processor from the other via doorbell registers in the bridge chip and interrupts may also be sent on completion of DMA transfers.

The Alpha processor runs no operating system or micro kernel which restricts code to a single thread. It is programmed in C and a library of synchronous and asynchronous communication functions are provided to facilitate parallel computing. Message passing between the Alpha and the rest of the network are performed by a server process on the Transputer. The DMA engines in the 9060 transfer data between the Alpha’s memory and the T9000’s local memory. The VCP in the T9000 sends and receives data on the DS links.

The results presented here (see Figure 8) demonstrate that data may be transferred from the Alpha’s memory and sent out on the DS links very efficiently. A program on the T9000 initiates a DMA transfer from the Alpha’s memory into buffers in an un-cached region of the T9000’s memory. This data is then transferred out of the TransAlpha via the DS links. Two buffers are defined so that the transfer of data from the Alpha memory into one buffer (via 70 Mbyte/s DMA transfer) may proceed concurrently with the transfer of data from the other buffer out onto one of the T9000 links.

In this way the bandwidth achieved on the single virtual link (4.64 Mbytes/s) is almost the same as that achieved sending directly from the T9000’s memory (4.74 Mbytes/s). With no overlapping of the DMA and the link communication the maximum achievable bandwidth would be 4.3 MBytes/s i.e. $\left(\frac{1}{70}+\frac{1}{4.7}\right)^{-1}$. It is also possible to drive multiple virtual links from the TransAlpha.

It is possible to read data directly from the Alpha’s memory and send it out over the DS links. This requires single word reads across the PCI bridge for which the effective bandwidth is 2.5 Mbytes/s. The overhead incurred by initiating a DMA transfer is 6.4 µs, but for messages greater than around 30 bytes in length the gain in bandwidth across the PCI bus makes DMA transfer the optimal mode.

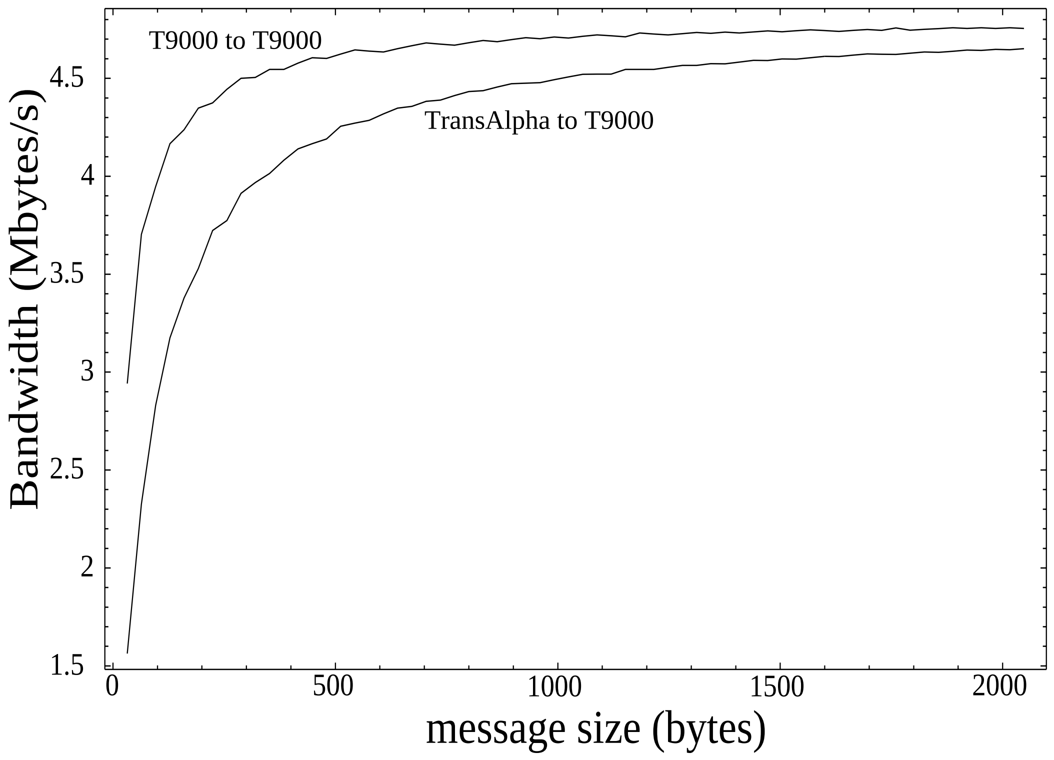

FIGURE 8 **Bandwidth versus message size for TransAlpha to T9000 uni-directional traffic over a single virtual link, connection via STC104.**

The message passing overhead between the TransAlpha and a T9000 (connected via a STC104) is 16 µs, the overhead between two T9000s connected via a switch is 9.6 µs. The extra time required for the TransAlpha is the 6.4 µs to initiate the DMA transfer.

A more detailed discussion of the TransAlpha module and its performance can be found in reference [7].

4 A PowerPC driving a DS link PMC

A basic DS link PCI Mezzanine Card (PMC) has been designed and built by DESY Zeuthen [8]. This card provides a DS link interface for embedded processors, for example, VME-based systems. The main components of the PMC are: the S5933 PCI controller from AMCC[9]; two 16 Kbyte FIFOs; the STC101 parallel to Data-Strobe-Link (DS link) converter[10]; a MACH 445 by AMD. A schematic of the PMC is shown in Figure 9.

The VCP functionality of the T9000 must be emulated by the host CPU, i.e. the PowerPC. The T9000 has been replaced by the STC101 link interface and a software emulation of the VCP on the host processor.

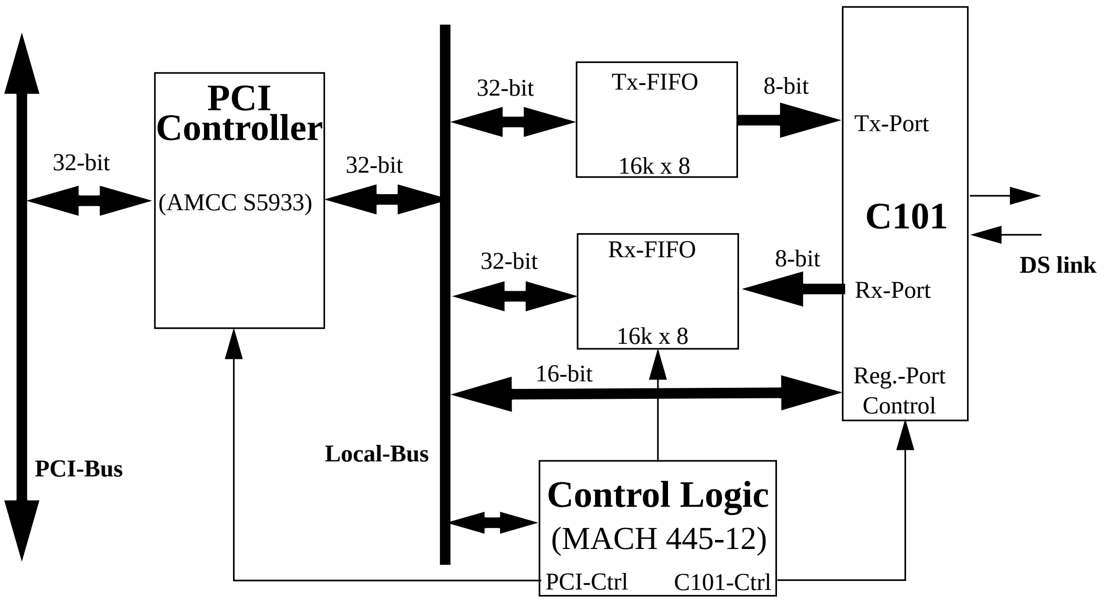

FIGURE 9 **Schematic of the DS link PMC.**

The PCI controller has both PCI target and bus master capabilities and has such features as mail box registers and DMA functionality. Two 16 Kbyte FIFOs connect the 32-bit local bus of the S5933 to the 8-bit Rx/Tx ports of the STC101 and allow PCI read/write burst accesses without wait states.

The STC101 is a full duplex link interface operating at up to 100 Mbits/s. It provides packetisation, framing and de-framing functionality. The packet size is programmable and with packetisation enabled the maximum data block length is 4 Kbytes. All registers of the STC101 can be accessed by single PCI read/write transactions via a 16-bit port.

A 100-pin Programmable Logic Device (PLD), the MACH 445-12, is responsible for the overall control and is programmable via a 10-pin JTAG port on the front panel of the PMC.

Software has been developed to enable an application running on a PowerPC processor embedded in a VME bus system to interface to the DS link. In particular, the software has been developed for the RIO2 806x[11]. The library emulates the VCP of the T9000 in software, implementing the same packet and message protocols as the VCP and allowing the multiplexing of multiple virtual links onto a single physical link. Figure 10 shows the bandwidth obtained between the PowerPC and a T9000 on a single virtual link, the maximum rate achieved is below 1.4 Mbytes/s and the CPU is totally saturated.

Data is passed between the PowerPC and the Rx/Tx FIFOs by DMA transfer. In order to avoid context switching overheads the STC101 does not interrupt the PowerPC on the arrival of every packet. Instead the PowerPC polls the STC101 registers with the disadvantage that a heavy load is placed on it's CPU. The current implementation does not use multiple virtual links concurrently.

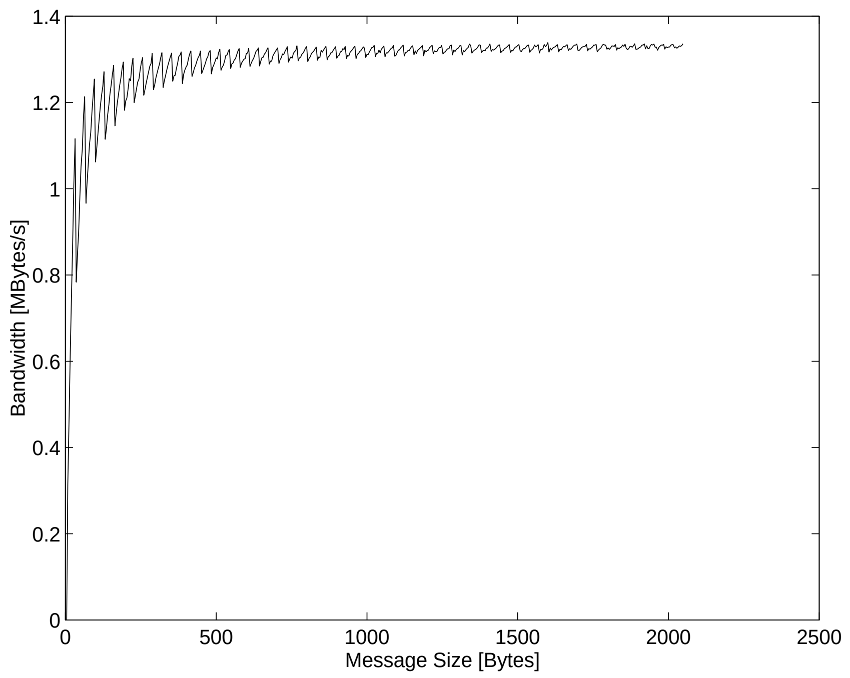

FIGURE 10 Bandwidth versus message size for PowerPC to T9000 uni-directional traffic over a single virtual link, connection via a STC104, packet size is 32 bytes.

The achieved bandwidth on a single virtual link between two directly connected PMC modules has also been measured (see Figure 11). The restriction of a maximum packet length of 32 bytes (imposed by the T9000) is now removed, so results for different packet lengths are shown. For comparison, the performance of two directly connected T9000s with a single virtual link reaches 6.4 Mbytes/s. The PowerPC implementation reaches this figure using 1024 byte packets, but for short messages (less than 1024 bytes) performance is greatly reduced. It should also be noted that the PowerPC is fully saturated for all measurements.

It is clear from the PowerPC results that to emulate the functionality of the VCP in software on the host processor does not give good communications performance and fully loads the CPU.

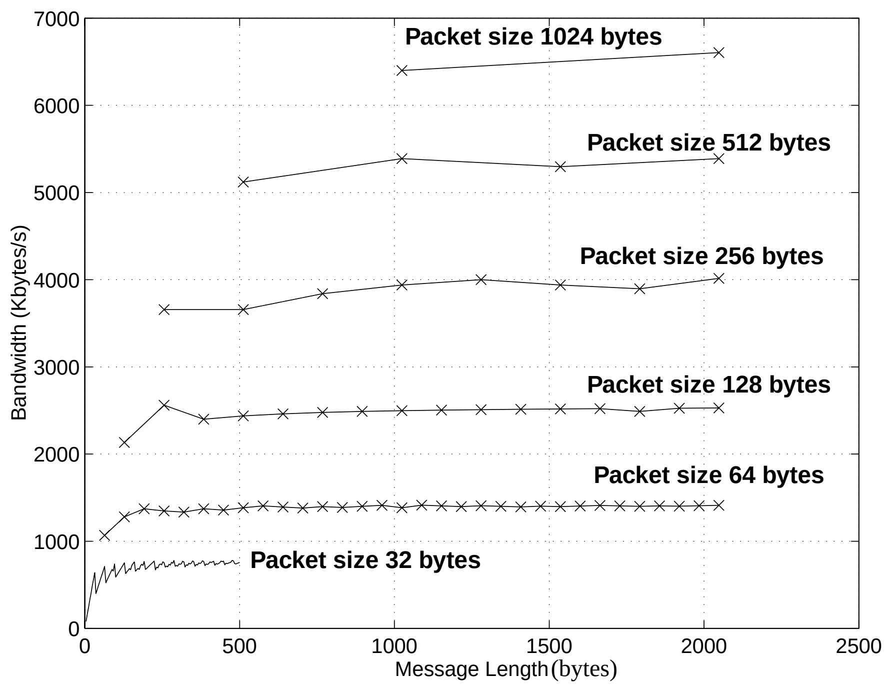

FIGURE 11 **Bandwidth versus message size for PowerPC to PowerPC (directly connected) over a single virtual link whilst varying the packet size**

5 Conclusions: what has been learned?

The TransAlpha module has shown that it is still possible to exploit the efficient communications performance of the T9000 when it is used as a communications co-processor. The throughput obtained is close to that for the T9000 stand-alone and there is a small increase in the message passing overhead.

The PowerPC implementation has shown that a software implementation of the VCP functionality puts a significant load on the host processor with inefficient communications performance.

Experience with the T9000 has shown the importance of low message passing overheads, high data throughput and low CPU loading.

Low message passing overheads are favoured by:

- An efficient interface between the user process and the network interface by using a dedicated communications processor (i.e. the VCP).
- Low process context switching times by using a hardware scheduler.

High data throughput is favoured by:

- Sufficient bandwidth between host memory and the network interface.
- Reducing or avoiding memory to memory copies.

- The use of multiple DMA channels to drive multiple virtual links over multiple physical links.

Low host CPU loading is favoured by:

- Reducing or avoiding memory to memory copies.
- Low interrupt overheads.
- Light weight protocols implemented in the VCP.

The T9000 has performed well in these areas, with the exception of a poorly designed external memory interface and a low clock speed. In addition, the acknowledge scheme of the T9000 restricts the per virtual link throughput. The above issues are addressed on a commodity computing platform in part 3 using DS link interface hardware described in part 2.

References

[1] IEEE Std. 1355, *Standard for Heterogeneous Inter-Connect (HIC). Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction*. IEEE Inc., 1995.

[2] *The T9000 Transputer Hardware Reference Manual*. Inmos Ltd., Inmos document number 72 TRN 238 01.

[3] R. Heeley, *Real Time HEP Applications using T9000 Transputers, Links and Switches*, Ph.D. Thesis, The University of Liverpool, October 1996.

[4] S.Fisher, *Low Level Benchmarking of the T9000 Transputer*, M.Sc dissertation, the University of Liverpool, February 1995.

[5] *The STC104 Asynchronous Packet Switch*, Data Sheet, April 1995, SGS-Thomson Microelectronics.

[6] Networks, Routers and Transputers, edited by M.D. May, P.W. Thomson and P.H. Welch, pp. 90, ISBN 90 5199 129 0.

[7] T.C. Carden, R.W. Dobinson, S. Fisher, P.D. Maley, High Performance Computing Nodes for Real Time Parallel Applications, Nuclear Instruments and Methods in Physics Research A, 394 (1997) 211-218.

[8] For more information contact K-H Sulanke, DESY Zeuthen, Platanenallee 6, 15738 Zeuthen. Tel. +49-(0) 33762-77207.

[9] *S5930-S5933 PCI Controllers*, Applied Micro Circuits Corporation.

[10] *The STC101 Parallel DS Link Adapter*, SGS-Thomson Microelectronics, June 1994.

[11] *RIO2 8061 and RIO2 8062 PowerPC based RISC I/O Board*, Technical Manuals, CES.

D 3.1.1 Report on interfacing Commodity processors to IEEE1355 DS links

PART 2: The architecture of the hardware interfaces

1 Introduction

The hardware platforms in this task were developed to provide access to the widest possible market for DS link technology. This was done by adopting the PCI standard to address both PC and workstation markets and its mezzanine equivalent PMC for the embedded systems markets of VME and Compact-PCI. This part of the deliverable describes the design choices and implementation of the two platforms.

2 Design Issues

2.1 The Bus

It was decided to exploit the large market share of the PCI bus to facilitate the connection of heterogeneous processors via DS links. PCI in its original form is the interface of choice for the IBM PC compatible platforms for commodity processors. Motherboards are available with processors from: Intel, AMD, Cyrix, DEC, Motorola etc.

The industrial market also profits from the PCI albeit in a different mechanical form factor, the PMC mezzanine bus. PMC mezzanine sockets are provided on many VME and COMPACT-PCI processor modules and used for standard or application specific I/O. These CPU modules are available from many different manufacturers and employ a variety of processors such as Intel, SPARC, Power PC, ARM etc.

2.2 Bus Interface

Having chosen the bus it was decided to employ a known and existing interface chip to limit the risks. The choice was made to use an AMCC design. This was a second generation component whose known bug list did not compromise any of the required design functionality.

2.3 Serial Link Interface

The STC101 component was the only available DS link interface device available.

2.4 Glue Logic

For compute systems doing heavy I/O the studies in Part 1 showed that the system performance is enhanced by using lightweight protocols and moving functionality away from the host CPU towards the I/O interface itself. The full T9000 I/O functionality could not possibly be added on to a processor having a different architecture due to the crucial role of the hardware scheduler.

This leaves a sliding scale of options available to the network interface designer. The cheapest and most inefficient approach uses a host based ‘software only’ solution. The inefficiencies involved however would make it difficult to find a market for such a solution. The most performant (and expensive) approach would be an ASIC that incorporated fixed logic for time critical operations together with programmable processing for flexibility. The cost of this is clearly beyond consideration.

What is sought then is some point at which sufficient performance can be obtained from the interface while remaining within a marketable cost.

It was decided then not to incorporate a communications processor but to see to what extent lightweight protocols could be accommodated using Field Programmable Gate

Array (FPGA) technology. Such devices are available over a wide range of capacity, speed and cost. At the time of design, devices were available with from 30,000 to 50,000 gate capacities, routed gate delays from 27nS down to 16 nS and costs from $80 to $480

During the architectural design phase there were requests for protocol specific operations such as packet concatenation and swinging buffers. Fast memory was added to the FPGA to cater for these functions.

The final architecture then provided for a range of price/performance options that would fulfill the requirements of a performant DS link network interface. We chose the largest such device available in order to explore the flexibility that this offered the design process. Just what the attainable performance limits were, is a subject of the research described in Part 3.

2.5 Platform Design

A PCI based design meeting the above requirements had been done independently of the ARCHES project as a joint collaboration between CERN and the INFN (Rome). This platform, named the DS Network Interface Card (DSNIC) was used in Arches to develop the software and firmware required for a PCI based host to exploit the message passing capabilities of DS links.

To meet the requirements of the embedded market a second interface board based on PMC mechanics was designed and built within Arches using the same approach as that employed in the DSNIC. The mechanical and power constraints of the PMC form factor required a few design changes, notably the use of only one DS link. Firmware addressing just one link can be used on both boards with only a simple re-compilation to account for the different packaging pin-outs.

3 DSNIC Architecture

The architecture is shown in Fig 1.

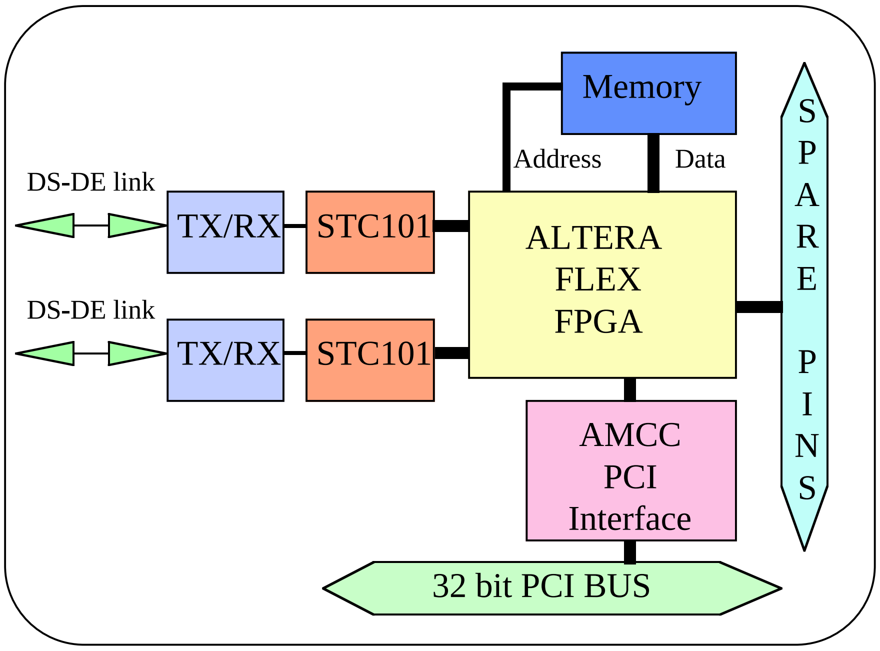

Fig 1. THE DSNIC Architecture Block Diagram

3.1 Design Details

The design goal was first to provide host control to the platform through some known interface chip (for security) and secondly to provide outputs in the form of one or two DS links. For simplicity there would be no embedded processor. All the intelligence, control, and ‘glue’ functionality would be subsumed into one large programmable gate array. This approach provided for a large degree of flexibility with the possibility of maximum hardware performance provided that the resources could be sufficiently exploited. It also ensured a simple hardware design and minimised design and development risk. The functionality and performance of the platform is therefore totally under the control, and limits, of the firmware loaded into the FPGA.

3.2 PCI interface

A commodity interface chip, the AMCC 5933, provides the necessary PCI master, slave, DMA, interrupt and mailbox facilities between the onboard resources and any PCI bus host. Details of the chip can be found at:
http://www.amcc.com/Products/PCI/S5933.htm
All the ‘user-side’ data and control lines are handled by logic implemented in the FPGA.

3.3 DS-LINK Interface

There are optionally two interfaces to DS links using the STC101 Parallel to Serial link drivers. Details of the chip can be found at:
http://www.hensa.ac.uk/parallel/vendors/inmos/ieee-hic/data/C101-04.ps.gz
The appropriate differential transceivers and common-mode rejection chokes are incorporated on-board to provide a secure physical layer for data transmission. Access to the STC101’s is entirely controlled from logic implemented in the FPGA.

3.4 Memory

A bank of 128Kbyte of SRAM is optionally available and configured as byte writeable, byte readable 64 bits wide by 16Kwords deep. Access to the memory is entirely controlled from logic implemented in the FPGA.

3.5 FPGA

All the onboard resources are controlled from the Altera FLEX 10K50 family in PGA packaging. Details of the chip specifications can be found at:
http://www.Altera.com/html/products/f10k.html
There is sufficient board space to permit the mounting of a ZIF socket for easy speed upgrades. The FPGA can be optionally initialised from:

- Serial Eprom on power up
- Parallel Eprom on power up.
- Parallel Byte Blaster ™ plug
- Software control over PCI bus (requires three wire patch to pcb)

3.6 Clocking

The PCI interface and the FPGA run at 33MHz from the bus clock. The STC101’s run at a nominal 50MHz.

3.7 Serial Memory

There are separate serial memories to initialise the BIOS variables of the PCI interface and the initial conditions of the Altera FPGA.

3.8 Spare pins

The FPGA has 30 spare pins if both STC101's are mounted and 130 spare pins if the second C101 is not mounted. All these spare pins are brought out to headers for exploitation in the patch area of user defined functionality. The intention here was to provide connectivity for a mezzanine processor board should the application need arise.

3.9 Mechanics

The electronics are mounted on a full-length PCI form factor board. Users should be aware that on some PC compatible motherboards this restricts the freedom of choice of available PCI slots since the long board may be in conflict with motherboard elements such as heatsinks, fans, memory subsystems etc.

A picture of the mounted and working board (without a front panel) is shown in Fig 2.

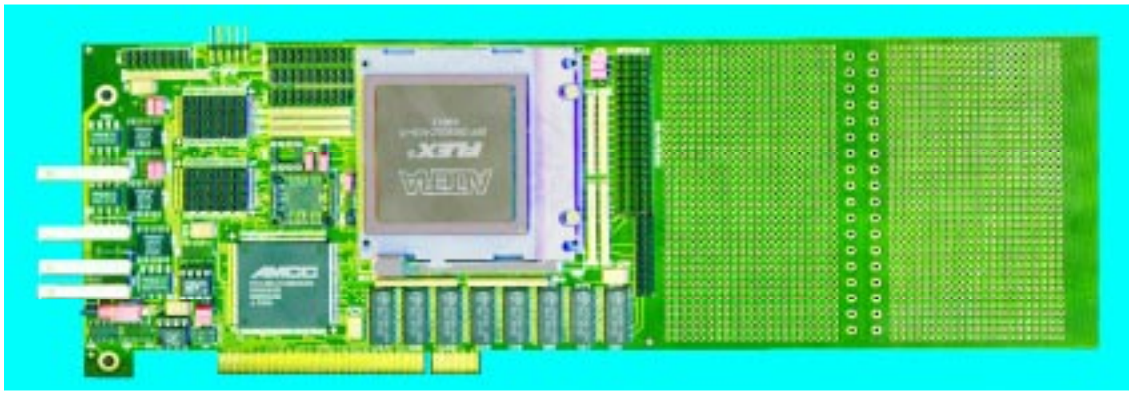

Fig 2 DSNIC photograph

4 DSPMC Architecture

The architecture of the DSPMC, very similar to the DSNIC, is shown in Fig 3. This common approach permits the easy migration of software and firmware from one platform to the other.

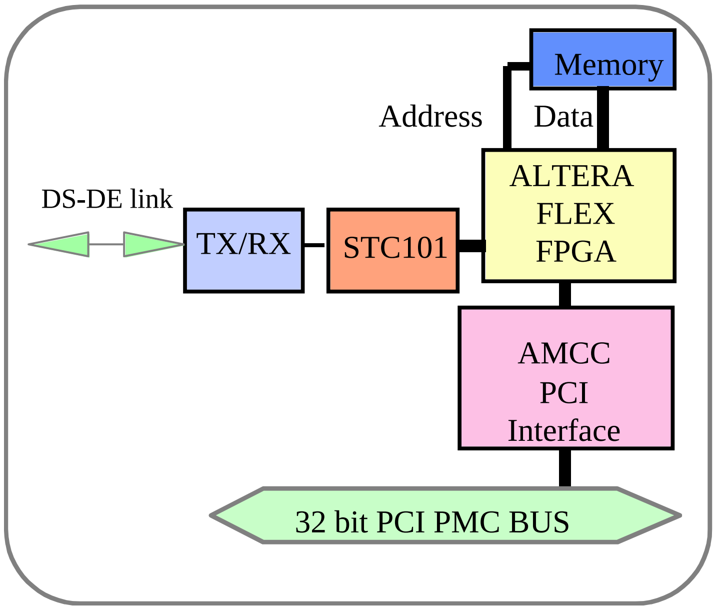

Fig 3 DSPMC Architecture Block Diagram

4.1 PCI interface

A commodity interface chip, the AMCC 5933, provides the necessary PCI master, slave, DMA, interrupt and mailbox facilities between the onboard resources and any PCI bus host. Details of the chip can be found at:
http://www.amcc.com/Products/PCI/S5933.htm
All the ‘user-side’ data and control lines are handled by logic implemented in the FPGA.

4.2 DS-LINK Interface

There is one DS link interface the STC101 Parallel to Serial link driver. Details of the chip can be found at:
http://www.hensa.ac.uk/parallel/vendors/inmos/ieee-hic/data/C101-04.ps.gz
The appropriate differential transceivers and common-mode rejection chokes are incorporated on-board to provide a secure physical layer for data transmission. Access to the STC101 is entirely controlled from logic implemented in the FPGA.
Given the uncertainties of continued support for the STC101, pcb tracking has been provided from the FPGA to solder pads at the serial input and outputs of the STC101. It would thus be possible to subsume the functionality of the STC101 into the FPGA in the case that the chip was no longer available.

4.3 Memory

A bank of 128Kbyte of SRAM is optionally available and configured as byte writeable, byte readable 64 bits wide by 16Kwords deep. Access to the memory is entirely controlled from logic implemented in the FPGA.

4.4 FPGA

All the onboard resources are controlled from the Altera FLEX 10K50 family in BGA packaging. Details of the chip specifications can be found at:
http://www.Altera.com/html/products/f10k.html
The FPGA can be initialised under software control over the PCI bus.

4.5 Clocking

The PCI interface and the FPGA run at 33MHz from the bus clock. The STC101 runs at a nominal 50MHz.

4.6 Serial Memory

There is a serial memory to initialise the BIOS variables of the PCI.

4.7 Spare pins

Some of the spare pins of the FPGA are brought out to test-point pads to assist in any needed firmware debugging or monitoring.

4.8 Mechanics

The components are mounted on a PMC double width board. A photograph of the DSPMC can be seen in Fig.4.

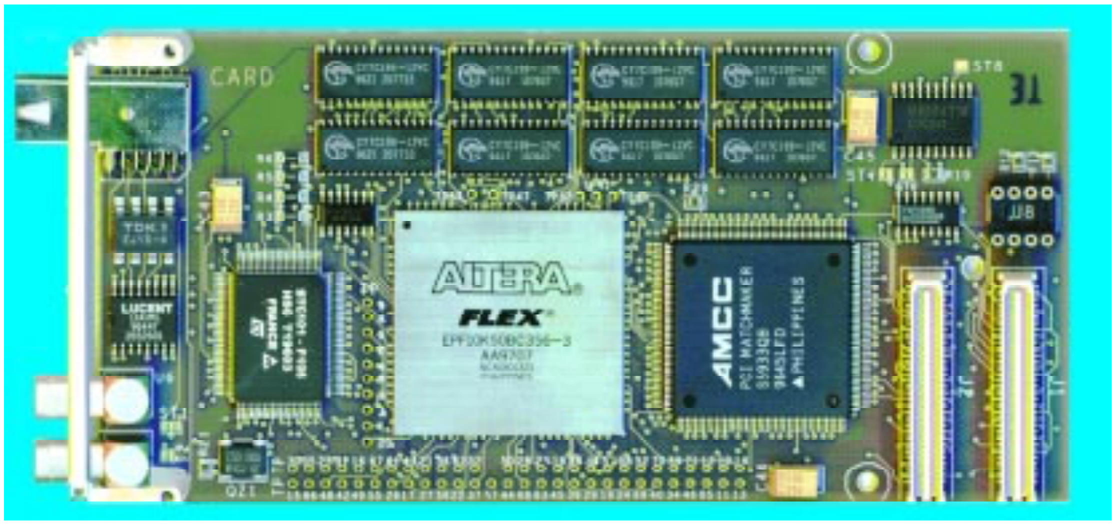

Fig 4 DSPMC photograph

D 3.1.1 Report on interfacing commodity processors to IEEE 1355 DS links

PART 3: The design and implementation of the software and firmware

Contents

1 Introduction

In part 1 we examined the baseline performance of DS links and identified the elements needed for their sucessfull exploitation. In part 2 we considered the hardware possibilities and limitations to construct cost effective yet flexible network interface cards that could attain attractive levels of throughput and latency. This part of the deliverable addresses the questions of which communication paradigms and protocols can best exploit the available resources and how best to partition the functionality between software in the host processor and firmware in the interface. The API and protocol design is explained and was implemented. Measured results are presented and discussed.

2 Switched IEEE 1355 DS link networks

IEEE 1355 DS links are bidirectional flow-controlled point-to-point serial links running at 100 Mbaud. DS link networks can be built using the 32 port SGS Thompson C104 packet switch[17]. This chip uses wormhole routing to route DS link packets; preloaded routing tables determine the packet's destination port on the basis of the packet header, as soon as this header has been received by the source port. The C104 does *not* use a bus based architecture, instead a non-blocking crossbar is implemented to route packets. It has a switching latency of 1 μs.

As part of the Macramé Esprit project, CERN has constructed a very large network testbed[21] based on DS links and C104 switches. This has allowed the investigation of latency and throughput for many different traffic patterns and network topologies up to 1024 terminal-nodes.

3 DSNIC board

A PCI based DSNIC board[3] has been developed jointly between CERN and INFN[11]. The wide acceptance of the PCI bus standard allows the board to support many present and future processors.

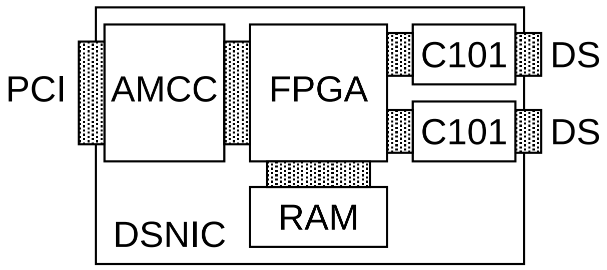

Figure 1: The DSNIC board.

The board, see Figure 1, contains the AMCC 5933 PCI interface chip, two SGS Thompson C101 parallel to serial DS link interfaces[16], 256 Kbytes RAM, and the Altera FLEX 10K50 Field-Programmable Gate Array (FPGA). The spare pins of the FPGA are brought out to headers to allow hardware extensions, e.g., via a daughter board. The flexibility of the FPGA allows part of the communication functionality to be off-loaded to the board; this functionality is thus spread over the board's firmware and the controlling software, i.e., the driver, executed by the host CPU.

Table 1: The communication API.

Basic CSP constructs
int **Send**(*NI** ni,*int* vl,*char** address,*int* length)
int **Receive**(*NI** ni,*int* vl,*char** address)
int **WaitAndSelectFirst**(*NI** ni,*int** vl_nrs,*int* size)
int **WaitAndSelectRandom**(*NI** ni,*int** vl_nrs,*int* size)
int **SelectFirst**(*NI** ni,*int** vl_nrs,*int* size)
int **SelectRandom**(*NI** ni,*int** vl_nrs,*int* size)
Non blocking communication
int **StartSend**(*NI** ni,*int* vl,*char** address,*int* length,*Request** r)
int **StartReceive**(*NI** ni,*int* vl,*char** address,*Request** r)
int **Completed**(*NI** ni,*Request** r)
int **Complete**(*NI** ni,*Request** r)
Asynchronous communication and its non blocking variant
int **ASend**(*NI** ni,*int* vl,*char** address,*int* length)
int **StartASend**(*NI** ni,*int* vl,*char** address,*int* length,*Request** r)
Message memory handling
*void** **Allocate**(*NI** ni,*int* size)
void **Free**(*NI** ni,*void** memory)
Initialisation and termination
int **Open**(*NI** ni,*char** resource)
void **Close**(*NI** ni)
void **SetUpVirtualLink**(*NI** ni,*int* nr,*int* buffer_size, *dword* remote_header,*int* remote_header_size, *int* remote_nr,*int* remote_buffer_size)

4 Choice of platform

We use high-performance commodity processors, like the DEC Alpha, Power PC, or Pentium, to perform real-time analysis because of their very good price/performance ratio. For the same reason, we use the Linux Operating System (OS): it is freely available, is highly reliable, and offers POSIX compatible real-time features. Our platform consists of 200 MHz Pentium Pro PCs running the Linux 2.0.27 OS.

5 Communication API

We base our parallel computer on Communicating Sequential Processes [8] (CSP). We use OS processes, i.e., Linux processes or threads, for the CSP processes. The CSP channels are virtual links between these processes. The Application Programming Interface (API), see Table 1, provides full CSP communication functionality, including facilities for active channel selection.

We add non-blocking communication to the blocking CSP interface, since non-blocking communication allows concurrent computation and communication without task switching. This improves the performance of the DSNIC, since task switching is a time consuming operation in general purpose OSs. The definition of blocking and non-blocking corresponds to the definition in the Message Passing Interface[12] (MPI) standard. In non-blocking communication, the send or receive call may return before the operation is completed, and before the user is allowed to re-use resources, such as buffers, specified in the call. In blocking com-

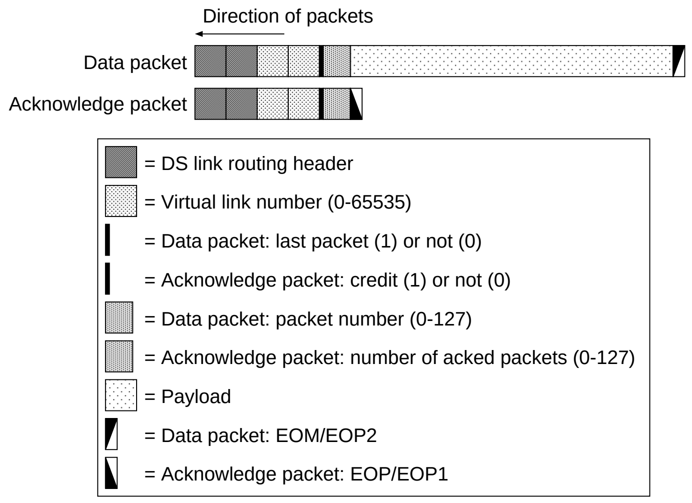
Direction of packets
Data packet
Acknowledge packet
= DS link routing header
= Virtual link number (0-65535)
= Data packet: last packet (1) or not (0)
= Acknowledge packet: credit (1) or not (0)
= Data packet: packet number (0-127)
= Acknowledge packet: number of acked packets (0-127)
= Payload
= Data packet: EOM/EOP2
= Acknowledge packet: EOP/EOP1

Figure 2: The format of data and acknowledge packets.

munication, returning from a send or receive call indicates that the user is allowed to re-use resources specified in the call.

Furthermore, we add asynchronous communication to the synchronous CSP interface. In synchronous communication, the sender and receiver processes not only exchange a message, but they also synchronise during this communication: the send and receive operations only terminate after they both have been executed. In asynchronous communication, the send and receive operations do not necessarily synchronise: the send operation may terminate even before the receive operation has been executed, by exploiting the use of buffers. By using buffers on the receiving host, asynchronous communication may result in latency improvements, since the message data may already have arrived at the destination host even before it is requested by the receiving process. The API allows the user to specify the amount of buffer space on the receiving host for each virtual link.

To avoid memory-to-memory copying on the host CPU, we make the DSNIC collect and deposit the message data directly in user space. Direct user space access puts constraints on the kind of memory that can be used: the memory must be locked, word aligned, and accessible via Direct Memory Access (DMA). Therefore, the API is extended with special allocation and release functions for message memory.

6 Protocol design

The IEEE 1355 standard describes the communication protocol up to the packet level: a DS link packet consists of a destination address, payload, and a packet delimiter. At the application level, processes need to be able to exchange messages. The gap between these two interfaces must be covered by a communication protocol. We try to optimise the performance, i.e., network latency and throughput, induced by the protocol.

6.1 Splitting messages into packets

The length of a message is not limited; it is determined by the application process. Messages must be communicated by exchanging packets. Using large packets in wormhole routed networks results in a performance penalty in both network latency and available network throughput due to network congestion. The impact of the packet size on the available network throughput has been shown in [6]: the maximum network throughput in a 512 end-node

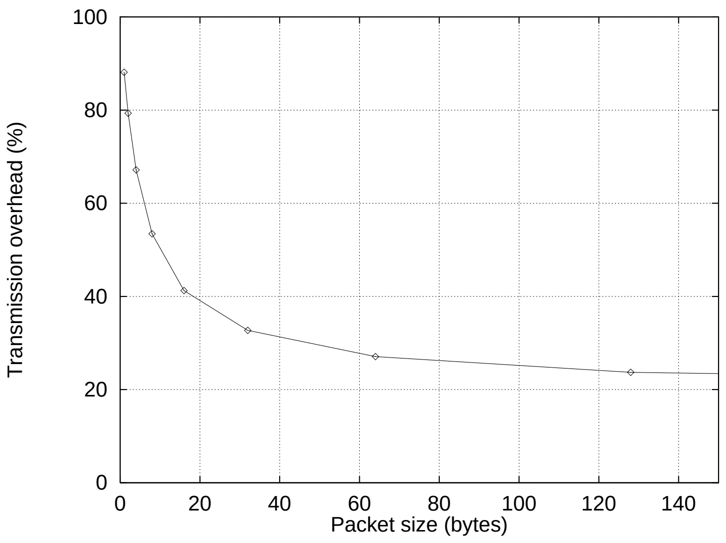

Figure 3: The transmission overhead against packet size for message length 1000.

Clos network drops by 22 % if packet size 1024 is used instead of 16. Section 6.4 presents more detailed network performance measurements that are related to the characteristics of the DSNIC protocol. To avoid the performance penalty which occurs if each message is transmitted as a single packet, we have decided to split up each message into a number of limited-size packets. Each message is sent as a number of maximum size packets, and one last packet, which is smaller than or as large as the maximum size.

6.2 Optimising throughput

The protocol supports the adaptive routing capabilities of the C104 switch to improve the network throughput performance[6]. However, adaptive routing can cause packets of the same message to arrive out-of-order at their destination.

The communication protocol of the T9000 transputer[18] uses a strict acknowledgement scheme to avoid out-of-order packet arrival. This scheme limits the maximum message throughput[7] due to the latency of the acknowledge. Measurements have shown that two T9000 transputers, interconnected via one C104, can achieve a throughput of 4.7 Mbytes/s over a single virtual link. Interconnecting them via five C104s, which increases the latency of the acknowledge by 4 μs, reduces the maximum throughput to 2.3 Mbytes/s.

We use a sliding window protocol[19], together with an acknowledgement scheme without the limit in maximum throughput: each acknowledgement packet can acknowledge a sequence of packets. The protocol does not use piggybacking because we do not expect a performance gain from it, since DS link networks perform well for small packets, and the protocol requires only a few acknowledgements per message.

6.3 Reliability and end-to-end flow control

The protocol should provide reliable communication. The Macramé testbed has proven that the per-link flow control, together with well designed hardware[13], can result in very reliable systems: a maximum Bit Error Rate (BER) of 9.6×10^{-18}[21] has been reported. We decide to accept the maximum BER of 9.6×10^{-18} for our purposes. The consequence is that the occurrence of an error in the network will result in unspecified system behaviour. We assume that the DS network does not to lose or corrupt any data, however, we should also avoid

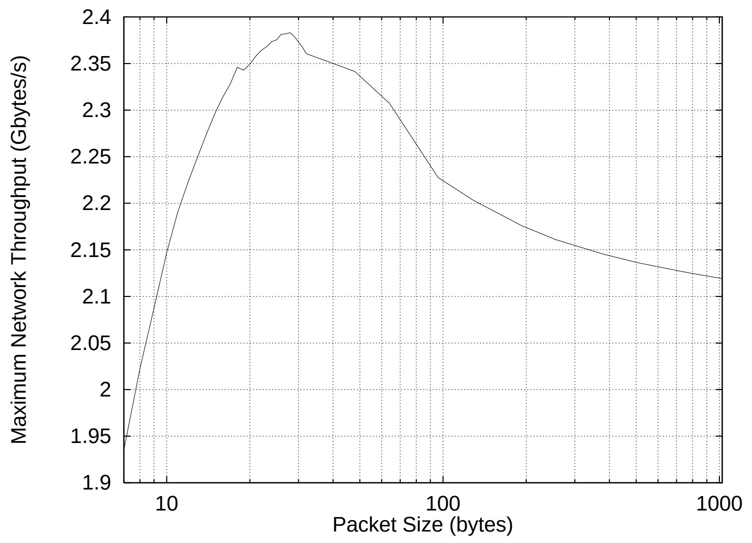

Figure 4: The maximum network throughput versus packet size.

data being lost by the message receiver due to overflowing buffers. Therefore, a means of end-to-end flow control is required.

We choose *not* to implement end-to-end flow control via the DS per-link flow control, holding data reception as soon as the buffers are full, since this blocks the network until the receiver executes a receive operation, and therefore leads to poor network performance and possibly deadlock. We use credit based end-to-end flow control: the receiver allows a sender to start message transmission by sending it a credit.

6.4 Transmission overhead and optimal packet size

We define the transmission overhead to be the number of bits, communicated for a message, that do not represent the data bits of the message. In particular, packet headers and acknowledge packets determine the transmission overhead. Most forms of serial communication use encoding, e.g., adding stop bits, to ensure that the receiving side correctly interprets the serial bit stream. DS links use an encoding scheme that extends each byte, 8 bits, to a 10 bit token. The transmission overhead is therefore at least 20 %.

Figure 2 shows the format of data and acknowledgement packets that we use for the DSNIC protocol. It shows that, apart from the DS link routing header, three characters are used for protocol specific information, independent of the payload size. Together with the acknowledgement scheme, this allows us to calculate the transmission overhead for different packet sizes. Figure 3 shows the transmission overhead against packet size, or, more precisely, against payload size, for sending a 1000 byte message. The packet size strongly influences the required number of packets and packet headers, and thereby the transmission overhead.

Figure 4 shows the influence of the packet size on the maximum network throughput for a 512 end-node Clos network under random traffic. This graph shows an optimal network throughput for packet size 28. For packets smaller than 28, the network throughput drops due to the domination of the transmission overhead. For packets larger than the optimum, the network throughput becomes worse due to network congestion.

The header of each packet needs to be processed by the DSNIC. This processing requires time. Using a small packet size, such as the optimal 28, requires a lot of processing to achieve full bandwidth communication. To keep this processing from becoming the system's

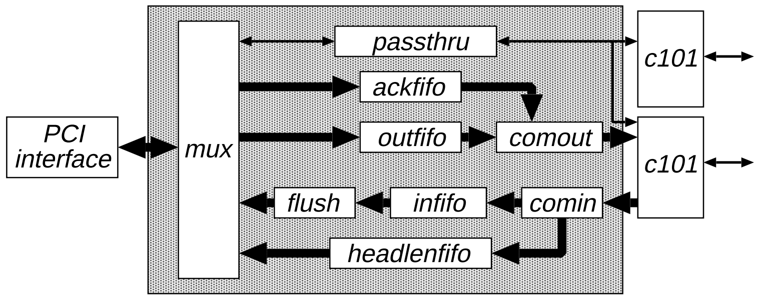

Figure 5: Block diagram of the firmware.

bottleneck, we choose not to fix the packet size, but to make it adaptable so that its influence on the performance of the DSNIC can be investigated. We only support powers of two for the packet size to accommodate the implementation.

7 System design

The DSNIC should offer the described API to processes, and implement communication via the described protocol. This functionality must be distributed over the FPGA firmware and the host software.

Measurements have proven that the maximum throughput cannot be reached if the host CPU is used to transfer every single byte of a message via register I/O. The firmware therefore minimally needs to offer a packet transfer interface. Being unable to estimate the development time and resource utilisation, i.e., the required number of Logic Cells (LCs), for complex firmware functionality, made us decided to initially only implement the packet transfer functionality in firmware.

A well-known problem in interfacing is memory-to-memory copying. Memory-to-memory copying is an expensive and often nonessential operation which causes CPU loading and latency. Consequently, we choose to avoid it: message data must be DMA-transfered directly to the right address in a process' memory.

7.1 Firmware

A block diagram of the firmware components is shown in Figure 5. Two main data flows can be recognised: data reception and data transmission.

Comin takes care of packet reception, it splits packet headers from packet data. The headers are sent to the *headlenfifo*, and the data is sent to the *infifo*. Apart from splitting, *comin* also counts the length of the packet, which is known as soon as the end-of-packet character has been received. This length is also sent to the *headlenfifo*. Data in the *headlenfifo* is delivered to an AMCC mailbox, which can generate an interrupt. On a header reception interrupt, the host CPU can determine destination address of the packet data and establish the receiving DMA, thereby avoiding memory-to-memory copying.

To avoid a store-and-forward mechanism, the receiving DMA must be established as soon as the header of the packet has been received. To set up a DMA, the length of the DMA transfer must be known beforehand. Therefore *flush* pads out each packet to the full packet size, even though less data is communicated over the DS link. DMAs can now be set up immediately on header reception, using the full packet size.

Comout takes care of sending data and acknowledge packets. If data is available in both the *ackfifo* and the *outfifo*, data in the *ackfifo* will be selected. This way, acknowledgements can bypass enqueued data packets.

In order to hide the reaction latency of the software driver, there are FIFOs on both the receiving and the transmitting side. The buffer size of 1 Kbyte was chosen because it fits well into the internal resources of the Altera FPGA. A 1 Kbyte FIFO can hide a reaction latency up to 100 μs, which should be sufficient.

Mux multiplexes the AMCC to all the FIFOs so that interleaved transmission and reception is possible. *Passthru* provides access to the registers of the C101s. The registers of both C101s are accessible via register I/O. The firmware only supports one C101 efficiently, i.e., with a DMA driven interface.

7.2 Host software

The host software consists of an interface library that provides the API, as shown in Table 1, to processes. This library communicates via system calls to a driver, resident in the kernel, that implements all the message communication functionality, using the packet transfer interface provided by the DSNIC board.

We choose to make the software interrupt driven and to deschedule processes performing blocking communication, in order to allow concurrent computation and communication.

The host software should take care of the communication of every packet, of every message, for any process, over any virtual link. These aspects make the host software complex. Using an Object-Oriented[15] approach helped in the development of this software.

By only using facilities that are common in every operating system that offers real-time facilities, such as process scheduling, memory locking, and interrupt handling, we facilitate portability of the DSNIC software to other operating systems with real-time features, assuming they allow access to a PCI bus. Furthermore, using only standard kernel facilities makes the DSNIC an extension of the OS: processes are *not* restricted in the use of other OS facilities, e.g., storage facilities or other communication facilities.

8 Measurements

We have used two benchmarks: Comms1 and Comms2. These benchmarks are based on Parkbench[9]. For both benchmarks, two Pentium Pro 200 MHz PCs, each running a communication process, are interconnected via the DSNIC with a single DS link. Not using a large DS network ensures that we are able to measure the process-to-process communication performance without network influence.

In both Comms1 and Comms2, the processes bounce messages of a certain length. For each message length, we obtain the process-to-process message latency by measuring the message bounce time and halving it. The throughput is obtained by dividing the message length by the message latency. In Comms1, the two processes, using blocking communication, bounce a single message. The link is therefore mainly used unidirectionally. In Comms2, the two processes exchange messages simultaneously, using both blocking and non-blocking communication. The link usage is therefore bidirectional. For an ideal bidirectional link, the maximum throughput in Comms2 is twice the maximum throughput in Comms1.

We have performed the Comms1 and Comms2 benchmarks for the following packet sizes: 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096. To ensure readability, we only show the results for packet sizes 8, 1024, and 4096 in the graphs. As a reference, we also show the benchmark results of the T9000 transputer. These results have been obtained on two 20 MHz T9000 transputers, that are interconnected via a C104 switch network. The communication

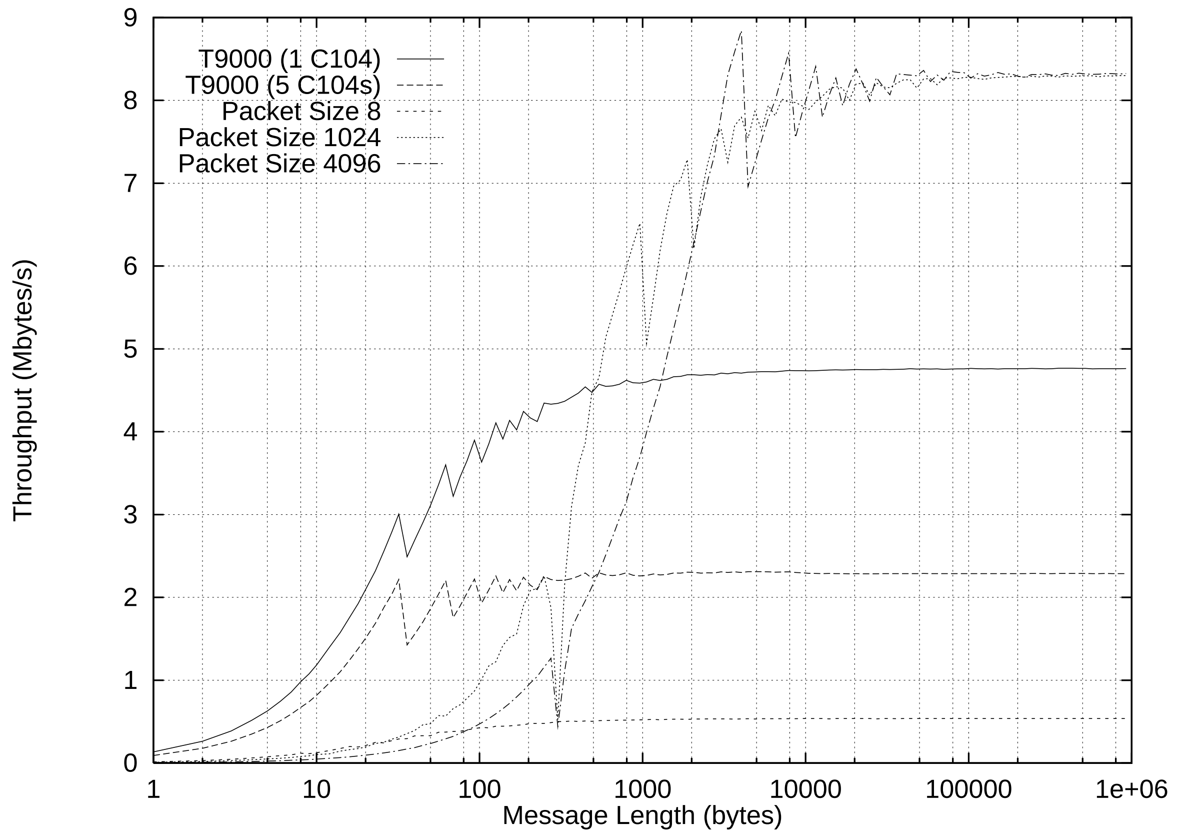

Figure 6: Throughput versus message length for Comms1 for various maximum size packets.

processes on the two T9000 transputers communicate via a single virtual link. The T9000 transputer uses a packet size of 32 bytes.

8.1 Throughput

Figure 6 shows the throughput versus message length for Comms1 for the three packet sizes and the T9000 transputer. The maximum throughput reached for long message lengths is 8.3 Mbytes/s. This is less than the maximum C101 throughput of 9.2 Mbytes/s. The reason for this is a limitation in the C101 receive FIFO. Only in some very specific cases, the limitation allows achieving a throughput higher than 8.3 Mbytes/s. The peaks in the 4096 packet size graph show such a situation.

The minimal packet size required to achieve the maximum throughput in Comms1 is 512. Using packet size 256 the 8.3 Mbytes/s is nearly reached. The other packet sizes are too small to be handled efficiently enough to reach the maximum throughput.

The plots in the graphs show a sawtooth pattern. This pattern is related to the packet size: if the message length exactly fits within a number of data packets, the packet handling overhead is relatively low, and therefore the throughput will show an optimum. If, however, the message length is one byte longer, an extra data packet is needed, so the packet handling overhead will be relatively high, which causes a throughput dip.

The maximum Comms1 throughput achieved on two T9000 transputer depends on the network latency, caused by the C104 switches that interconnect the two transputers. Directly interconnecting the T9000 transputers will result in an increase of maximum throughput of about 25 %. Due to the use of a different acknowledgement scheme, interconnecting two

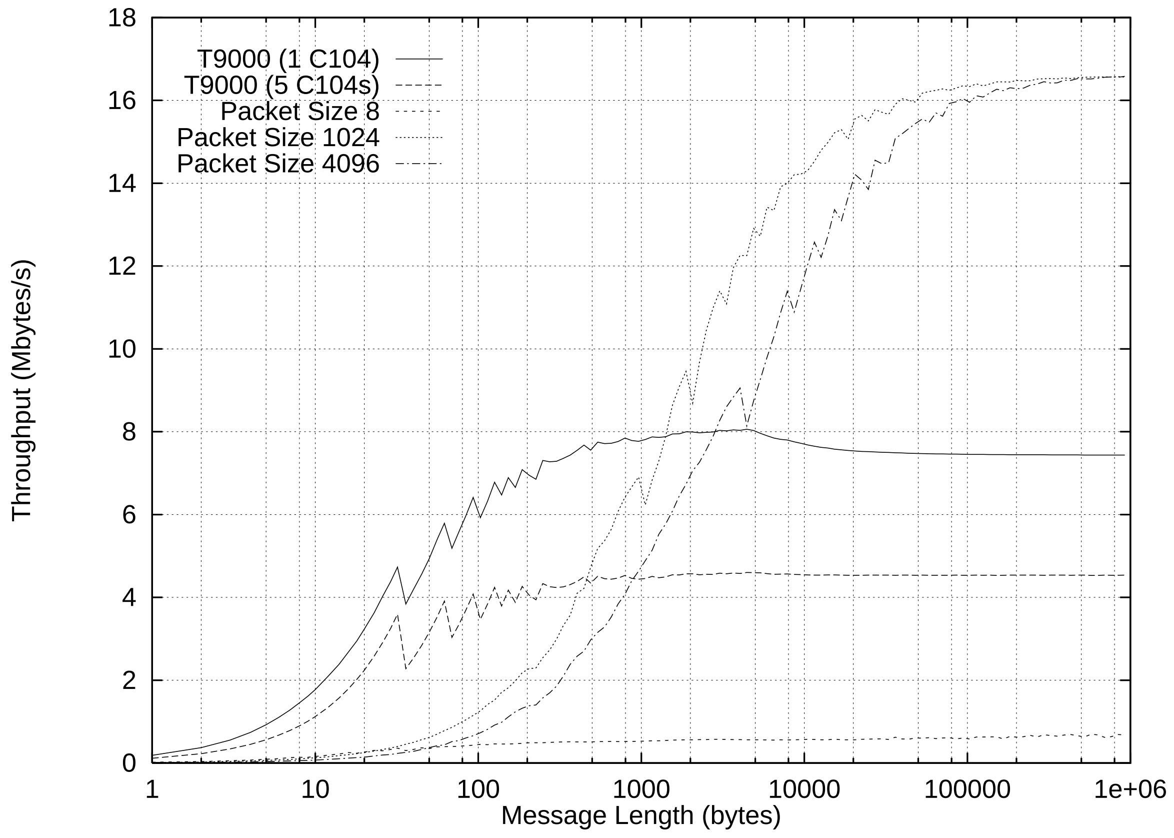

Figure 7: Throughput versus message length for Comms2 for various maximum size packets.

DSNICs via a number of C104s does not affect the maximum throughput, it only affects the latency.

Figure 7 shows the throughput versus message length for Comms2. As expected, the maximum throughput reached for long messages is 16.6 Mbytes/s: twice the 8.3 Mbytes/s Comms1 throughput. This proves that the DSNIC can fully exploit the bidirectional bandwidth of DS links.

The minimal packet size required to achieve the maximum throughput in Comms2 is 512. Packet size 256 definitely does not achieve the maximum throughput. Considering the Comms1 situation, this is to be expected: since the throughput is twice as high, the packet size must roughly be doubled in order to keep the CPU load the same.

8.2 Latency

Figure 8 shows the Comms1 latency versus message length. For message length 1, the graph shows the minimal latency. The minimal latency for all packet sizes up to 512 bytes is 67 μs. The minimal latency for 1024, 2048 and 4096 sized packets is higher. Similar results have been obtained for Comms2.

The higher latency for large packets, see Figure 5, is caused by the padding of *flush*. This overhead can be removed by modifying the protocol, so that the packet's header contains the packet's length. This change will make the minimal latency 67 μs, independent of the packet size.

The T9000 transputer has a minimal Comms1 latency of 7.4 μs; two directly connected T9000 transputers will even improve on this. The factor of 10 difference shows the per-

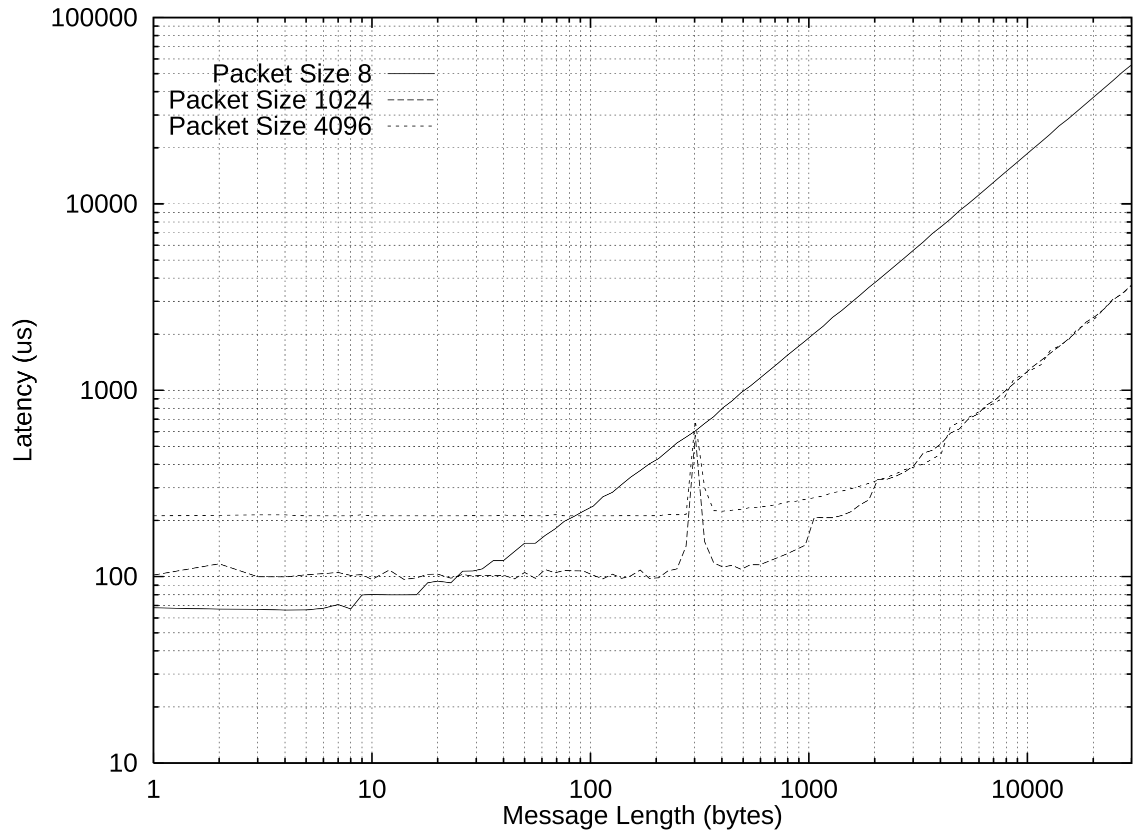

Figure 8: Latency versus message length for Comms1 for various maximum size packets.

formance lost due to the use of standard OS facilities, implemented in software, instead of dedicated hardware, i.e., the hardware communication system and the hardware scheduler.

8.3 Interrupt packet handling

In the DSNIC, an interrupt handler takes care of all the packet handling to communicate messages. Figure 9 shows for Comms1 the number of packets handled per interrupt versus message length. The results in this graph are strongly related to the way in which the interrupt handler operates. The interrupt handler is activated by a number of hardware events, e.g., FIFOs, or mailboxes, becoming full or empty. The interrupt handler could terminate as soon as the events that interrupted the processor are handled. However, in order to improve performance, the interrupt handler, after having handled the events, checks whether any other hardware events need handling. If so, it handles them, and checks again for new events. Therefore, the interrupt handler can handle many packets each time it is called.

Handling many events per interrupt improves performance, since it avoids unneccesary processor interrupts. However, if many events are available, the interrupt handler is active for a long time. During this time no other interrupts can be handled, which has proven to cause problems to the timer interrupt. To avoid interrupt handling that takes too long, we limit the maximum number of events handled per interrupt. The effect of this is shown by the limit of about 70 handled packets per interrupt for the smaller packet sizes.

In Figure 9 two stable states can be identified, *many* and *one*: the state *many*, handling the maximum number of packets per interrupt, and the state *one*, handling one packet per interrupt. The small packet sizes result in such a large event handling frequency compared

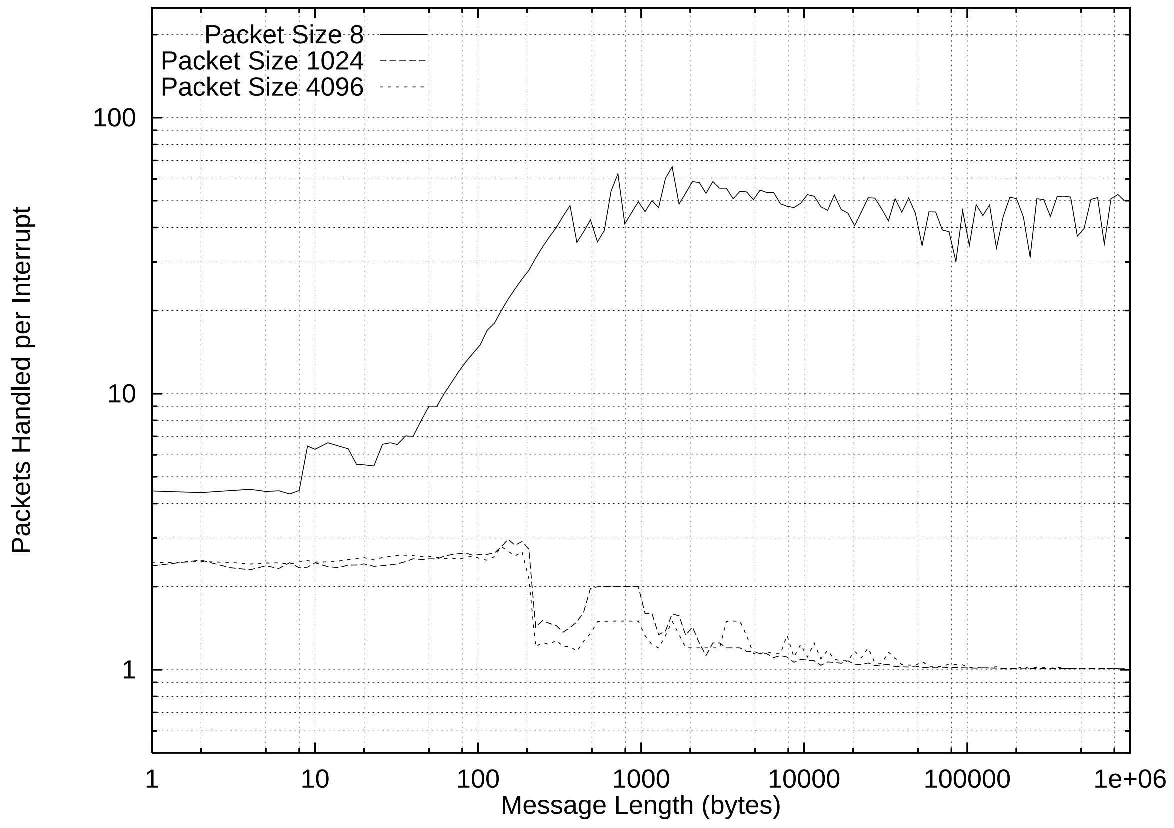

Figure 9: Number of packets handled per interrupt versus message length for Comms1 for various maximum size packets.

to the event handling time, that state *many* is reached. The large packet sizes result in such a low event handling frequency that state *one* is reached.

The larger packet sizes show a drop off at message length 270 to about 1 packet handled per interrupt. So, if a message of more than 270 bytes is sent, the interrupt handler only handles one packet, because at the end of the handler no new events are available. Handling one packet on interrupt therefore takes as much time as it takes to communicate a 270 byte packet. This drop in efficiency of the interrupt handler also causes the latency peak and throughput dip at message length 270, see Figure 8 and Figure 6.

Figure 10 shows the number of packets handled per interrupt versus message length for Comms2. In Comms2, we are dealing with bidirectional communication. Therefore, three stable states can be identified, the two states from Comms1, i.e., states *many* and *one*, and one extra state, state *two*. In state *two*, two packets are handled per interrupt, an outgoing and an incoming packet. Medium sized packets, such as packet size 1024, end up in state *two*.

8.4 Processor load

The communication results can only be put into perspective if the communication load to the processor is known. Consequently, we have also performed load measurements. The load is measured by concurrent execution of a communication task and a computation task on each of the two processors. The computation task measures the remaining processor power. It does this by increasing a counter in memory. By measuring the counter increase per second on an unloaded system beforehand, we can determine the processor load in a system with

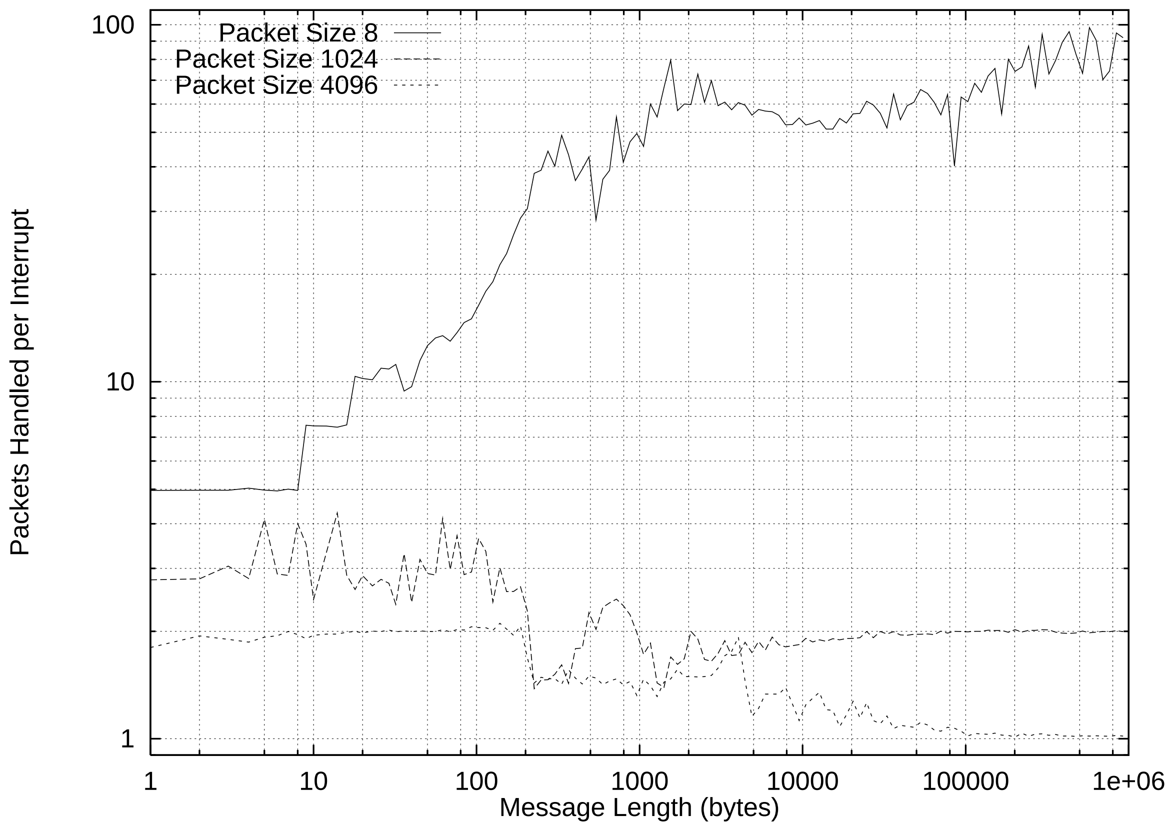

Figure 10: Number of packets handled per interrupt versus message length for Comms2.

communication.

In Linux, there are two facilities to execute a task: processes and threads. Each process has its own memory space, which keeps all other processes from accessing this memory. Threads exist within processes, sharing the memory space of the process. Memory spaces are mapped onto physical memory by the Memory Management Unit (MMU) of the processor. Context switching requires changing the mapping of the MMU. Since threads of the same process use the same mapping, context switching between these threads takes less CPU time than context switching between processes.

The amount of time available for a process depends on the amount of time assigned to it by the scheduler. To avoid suffering of the communication process from the calculation process, a high priority real-time scheduling strategy is used for the communication process. This strategy should make sure that the communication process gets scheduled-in as soon as work for it is available, i.e., at the moment the blocking communication has finished.

Figure 11 shows three Comms2 throughput graphs: without computation, computation by a thread, and computation by a process. The graphs show that using concurrent threads has little impact on the communication, and that using processes has a significant impact. We can estimate this impact by using a quantification based on the Comms1 and 2 linear timing model[9]:

$$t_n = t_0 + \frac{1}{r_\infty} \times n$$

$$r_n = \frac{n}{t_n}$$

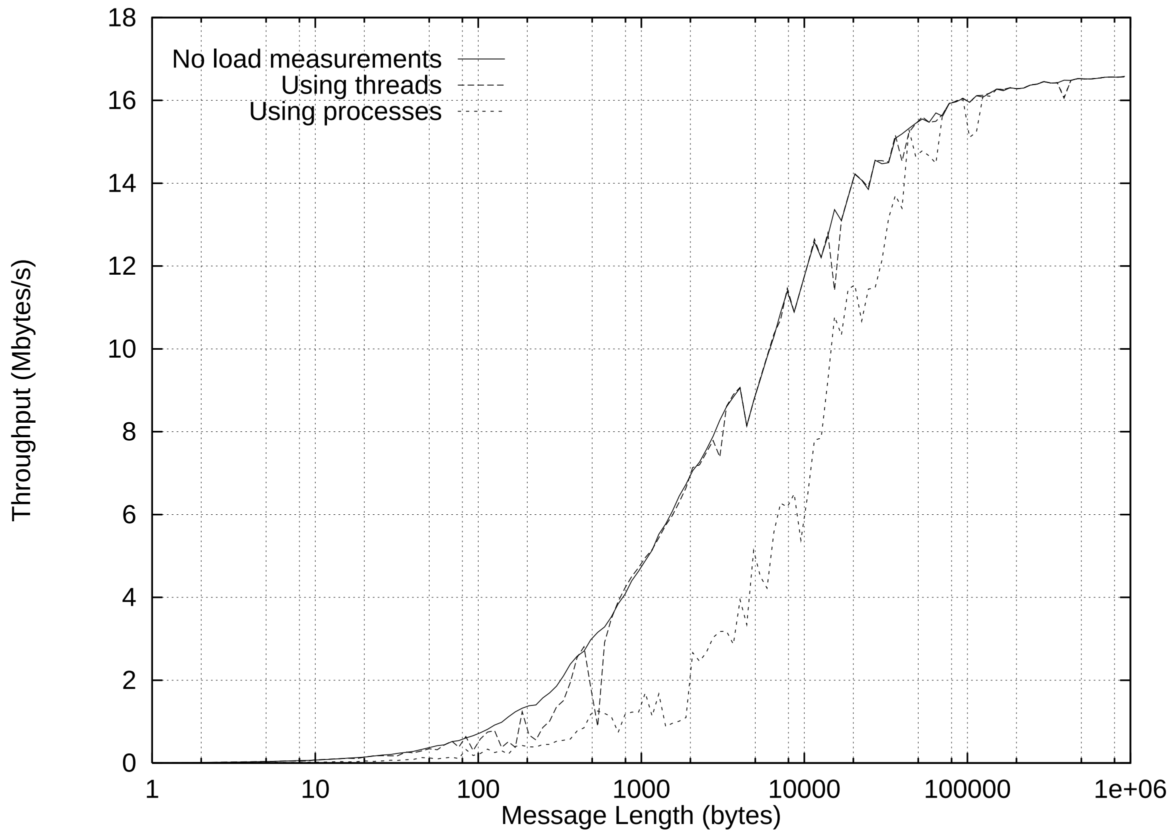

Figure 11: The effect of computation on the throughput for different message lengths, Comms2 and maximum packet size 4096.

In this model, t_n is the latency for a message of size n; t_0 is the latency to send a zero byte message; r_n is the throughput; and r_∞ is the asymptotic bandwidth, i.e., $\lim_{n\to\infty} r_n = r_\infty$, the limit on the maximum throughput. From this model one can derive that:

$$t_0 = \frac{n_{1/2}}{r_\infty}$$

The value $n_{1/2}$ is the half-performance message length, i.e., the message length required to achieve half the asymptotic bandwidth. Knowing $r_\infty = 16.6$ Mbytes/s, we can estimate t_0, by obtaining $n_{1/2}$ from the throughput graphs in Figure 11. These estimations, see Table 2, show that the context switching overhead due to the use of processes instead of threads is significant: a difference of about 600 μs in t_0. The T9000 results, which are also included in this table, show how the T9000 outperforms the DSNIC with respect to 0-byte message latency.

The CPU load of both the process and the thread version are similar for long messages. This is to be expected: for large messages few process switches need to be performed because the communication task is nearly continuously sleeping, so the load is caused by interrupt handling which is identical in both situations. In that case, the interrupt frequency, which is directly related to the packet size, has a major effect on the load.

Table 3 shows the maximum CPU load measured during Comms1 and Comms2 for 0.5 Mb up to 1 Mb messages. For Comms1, we see that the CPU load roughly doubles if the packet size halves, so *packet size* $\times$ *load* $=$ *constant*. According to this formula, the *constant* is the packet size at which the CPU load would be 100 % to achieve full bandwidth

Table 2: Estimations of the 0-byte message latency t_0, for Comms2 and packet size 4096.

Situation	$n_{1/2}$**(bytes)**	t_0**(μs)**
No load	3000	187
Thread load	3000	187
Process load	13000	783
T9000 (1 C104)	26	3.25

Table 3: Maximum CPU load for long messages.

Packet size (bytes)	**Comms1 CPU load (%)**	**Comms2 CPU load (%)**
8	100.0	92.8
16	100.0	87.4
32	100.0	82.8
64	99.9	79.5
128	99.9	89.8
256	93.7	93.0
512	55.3	97.2
1024	28.9	53.4
2048	14.1	36.1
4096	7.1	19.0

utilisation. Using packet sizes smaller than the *constant* will not allow full bandwidth utilisation. For Comms1, the *constant* is approximately 290 for packet sizes of 512 and more. The 290 indicates that the maximum throughput cannot be reached with packet size 256, as we have seen before. Furthermore, the 290 corresponds to the end of the throughput dip in Comms1, where the expected throughput is reached again, see Figure 6.

9 Analysis

9.1 CPU time for handling a packet

During the communication of small messages, the CPU is almost completely dedicated to packet handling: it is nearly continuously executing the protocol for every packet that is communicated. The only interference is caused by exiting the interrupt handler for a short while when the maximum number of events that can be handled in a single interrupt has been reached. This allows us to calculate a maximum of CPU time required to handle a packet if it would be dedicated to this task. For Comms2 and a packet size of 8 bytes, the maximum throughput reached is 0.696 Mbytes/s. This means that the CPU, if dedicated to packet handling, can handle a packet within 11.5 μs. Since the same protocol implementation is used for any packet size, the required CPU time for handling a packet does not depend on the packet size.

9.2 Off-loading to a dedicated processor

A dedicated processor might be an interesting solution for off-loading more DSNIC functionality to the board. The CPU time for handling a packet allows us to relate the CPU power of such a dedicated processor to the packet size that can be handled by this processor, assuming the same communication protocol is used. This would result in a system in which the board takes care of all the packet handling, and the interface between the CPU and the board allows message communication.

Suppose the dedicated on-board processor is n times slower than the 200 MHz Pentium Pro. This means that it requires $n \times 11.5$ μs to handle a packet. To be able to keep up with the full bidirectional DS link bandwidth, 16.6 Mb/s in our case, we can calculate the required minimal packet size: 16.6 Mbytes/s $\times$ 11.5 μs $\times$ n.

The typical message length used for real-time analysis in ATLAS is 1 kbyte. For ATLAS, the dedicated processor is therefore required to be capable of handling packets of this size adequately. A communication processor that is 5.4 times slower than the Pentium should just be capable of link saturation. It is expensive to use a dedicated processor to achieve the optimal packet size of 28 bytes, since it requires a processor with about 7 times the power of the Pentium. If such a small packet size is required, one has to use dedicated hardware.

9.3 Interrupt service time

In Section 8.3, we derived that an interrupt takes as much time as it takes to send a 270 byte packet. Knowing that data is sent at 8.3 Mbytes/s, we can derive that it takes 32.5 μs to service an interrupt.

This interrupt service time can be checked against the CPU load for long messages. For long messages, the processor load depends only minimally on the task switching time. In this case, the CPU load is caused by interrupts and memory bandwidth usage.

Consider the Comms1 communication of long messages in 4096 byte packets. Knowing that an interrupt takes 32.5 μs to handle, we can conclude that on an average, communicating a single byte requires 32.5 / 4096 = 7.93 ns CPU time. It requires 65800 μs CPU time every second to sustain the full data rate of 8.3 Mbytes/s. The CPU load due to interrupts for long messages and packet size 4096 is therefore 6.6 %. This 6.6 % is close to the 7.1 % observed. The difference can be explained by the memory bandwidth usage, and cache disturbance due to the interrupts.

9.4 Interrupt overhead

We have derived that a packet can be handled in 11.5 μs, and that an interrupt is serviced in 32.5 μs. Therefore, we can calculate the interrupt overhead for an interrupt that services one packet: 32.5 - 11.5 = 21 μs. The interrupt overhead is caused by context switching. On interrupt handler entry, the context of the current process and its thread must be saved. On exit, it must be restored. Another negative effect of context switching is cache usage. On handler entry, memory locations different from the memory locations in the cache are used, and therefore cache updates are required. In a similar way, the interrupted process suffers from the interrupt: the cache it was using is disturbed, and cache updates are required.

10 Performance improvements

Improving performance means obtaining higher throughput, lower latency, and lower CPU load during communication. The communication performance depends on both network and interface performance. Network performance is influenced by the communication protocol, as discussed in Section 6. The interface performance is determined by the efficiency of the implementation of the required operations.

We can distinguish three classes of operations: (1) *O*(*message length*) operations, such as copying of message data and CRC checking, (2) *O*(*nr of packets*) operations, the handling of packets, and (3) *O*(*nr of messages*), communication initialisation and task switching.

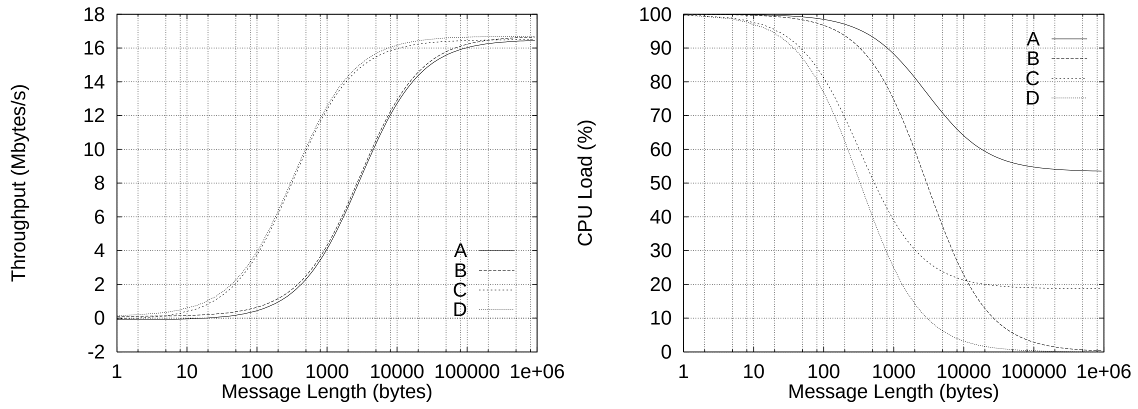

Figure 12: The effects of two performance improvement methods on the throughput and CPU load graph for Comms2 and packet size 1024. Graph A models the performance of the current system when using threads to perform the computation. Graph B models the performance of a system in which all packet handling is off-loaded to the DSNIC board. Graph C models the performance of a system with low-latency context switches. Graph D models a system that takes both the advantages of B and C. Compared to A and B, graphs C and D show an improvement in throughput.

By using DMA directly to process space and a reliable network, we have avoided implementing any *O*(*message length*) operations on the CPU. The maximum throughput is therefore limited by the C101 hardware and not by the CPU power. We have chosen to implement the *O*(*nr of packets*) operations, i.e., the packet handling, on an interrupt basis. The *O*(*nr of messages*) operations are implemented as kernel calls that execute reschedule operations.

We focus on two methods to improve performance: using the DSNIC board for packet handling, and using alternatives for OS functionality. By extending the Comms1 and 2 communication model so it takes the CPU load caused by communication into account, we can estimate the effect of the two performance improvements on the throughput and the CPU load graph, see Figure 12.

10.1 Modelling the CPU load

We model the CPU time to communicate a message of length n by the sum of a fixed time c_0 and a time dependent on the message length n:

$$c_n = c_0 + c_{\mathit{byte}} \times n$$

The factor c_{byte} is the per-byte CPU time. Notice that $0 \leq c_0 \leq t_0$. Knowing the CPU time to communicate a message, c_n, and the communication latency, t_n, the following equation expresses the CPU load l_n:

$$l_n = \frac{c_n}{t_n} \times 100\ \%$$

The value l_0, which equals $(c_0/t_0) \times 100$ %, expresses the CPU load for sending a 0-byte message. Furthermore, we can derive the following for the asymptotic CPU load l_∞:

$$l_\infty = \lim_{n\to\infty} l_n = c_{\mathit{byte}} \times r_\infty \times 100\ \%$$

In Section 9.3, we have already used this equation. The asymptotic CPU load, l_∞, calculated there, using $c_{\mathit{byte}} = 32.5$ μs$/4096$ bytes $= 7.93$ ns/byte for the per-byte CPU time for 4096 byte packets, matches the measured CPU load for long messages.

The results presented in this paper allow us to derive the parameters of the model that match the Comms2 communication using a packet size of 1024, i.e., the typical packet size required for ATLAS, and threads to perform the computation results. From Figure 7, we can determine that r_∞, the asymptotic bandwidth, equals 16.6 Mbytes/s. Furthermore, this figure allows us to determine that $n_{1/2} = 1480$. Since $t_0 = n_{1/2}/r_\infty$, we know that $t_0 = 178\ \mu$s. Knowing from Table 3 that $l_\infty = 53.4$ %, allows us to determine that $c_{\mathit{byte}} = 32.17$ ns/byte, using $c_{\mathit{byte}} = l_\infty/(100\ \% \times r_\infty)$. We cannot derive c_0 from the measurements, therefore we assume the CPU load is 100 % during 0-byte message communication, so $c_0 = t_0$. These parameters have been used to reconstruct the performance graphs of the current system, see graph A in Figure 12.

10.2 Using the DSNIC board for packet handling

By off-loading all the packet handling to the DSNIC board, as discussed in Section 9.2, all the packet handling interrupts, which are *O*(*nr of packets*), can be avoided. Avoiding these interrupts has no effect on the communication latency, since this latency is caused by the *O*(*nr of messages*) operations. Since latency and throughput are directly coupled in Comms1, there is no effect on the throughput graph. The gain is to be expected in CPU load. For ATLAS, which requires packet size 1 Kbyte, this can result in a performance gain of up to 53 %, see Table 3. This potential gain is modelled using $c_{\mathit{byte}} = 0$, see graph B in Figure 12.

10.3 Low-latency context switches

Context switches are the main cause of both latency and CPU load. We can distinguish three types of context switches: (1) rescheduling, requiring context switches between processes or between threads, (2) kernel calls, requiring context switching between process and kernel space, and (3) interrupts requiring context switching between process and kernel space.

In a general purpose OS, these context switches require a lot of CPU time, and thereby cause both latency and load. Using light weight alternatives for the OS context switches can therefore result in a significant performance improvement. However, a light weight alternative usually implies a restriction in some other field, e.g., no protection between processes, a limitation in the use of OS functionality, or support for only a single process. If the restriction is acceptable, this method offers an interesting solution. Solutions that apply this method can be found in [4], [14], and [20]. Graph C in Figure 12 shows the potential of such an improvement, assuming the 0-byte latency t_0 is 20 μs, instead of $187\ \mu$s, and assuming the interrupt handling overhead can be completely removed, which reduces the interrupt handling time from 32.5 μs to 11.5 μs. Furthermore, graph D in Figure 12 shows the effect of applying both the improvement methods: off-loading and low-latency context switches.

11 Conclusion

We have developed the DSNIC by continuously keeping the design aim in mind: optimise for low-latency high-throughput communication, requiring little CPU load. To accomplish this, the CSP based API is extended with facilities for asynchronous, non-blocking, and zero-copy communication. Furthermore, the communication protocol splits messages into limited size packets. This avoids continuous wormhole blocking and reduces the network latency. The protocol supports adaptive routing, and allows full bandwidth utilisation by a sliding window protocol and a non-restricting acknowledgement scheme. Reliability is ensured by the very low BER of DS link networks and the use of end-to-end flow control.

This way, we have managed to obtain a system that reaches 8.3 Mbytes/s unidirectional process-to-process throughput over a single virtual link. A maximum throughput of 16.6 Mbytes/s is reached for bidirectional communication. In both unidirectional and bidirectional communication, up to 90 % of the maximum theoretical bandwidth can be exploited. The DSNIC offers a 1-byte message latency of 67 μs. This is fast, considering that this is the process-to-process latency. During 8.3 Mbytes/s unidirectional communication, the DSNIC requires 7 % CPU load for a packet size of 4096 bytes. For that same packet size and 16.6 Mbytes/s bidirectional communication the CPU is loaded for 19 %.

We have observed that high throughput and low CPU load can only be obtained by using large packets, e.g., 4096 bytes long. On the other hand, we have shown that the ideal packet size for the 512 Clos DS link network is small: 28 bytes. Filling this gap completely would require dedicated hardware.

We have shown that protocol off-loading has the potential to result in a significant gain of available CPU power, e.g., up to 53 % for packet size 1024. Furthermore, we have shown that this method requires a substantial amount of on-board computing power.

In general purpose operating systems, context switches are a source of overhead. We have shown the overhead of one form of context switch: the interrupt. An interrupt that handles the communication of a single packet takes 32.5 μs, whereas the same operation on the same CPU can also be performed in 11.5 μs, an overhead of 65 %. Furthermore, we estimated the performance loss caused by context switches when using processes instead of threads to implement concurrent communication and computation.

The DSNIC has outperformed the T9000 in terms of throughput: the DSNIC achieves a maximum bidirectional throughput of 16.6 Mbytes/s compared to 8 Mbytes/s for the T9000. However, in terms of latency the T9000 outperforms the DSNIC: the latency for sending a single byte message on the T9000 is 7.4 μs compared to 67 μs for the DSNIC. This order of magnitude difference is due to the T9000's use of on-chip dedicated hardware to perform scheduling and communication. In contrast these functionalities are mainly provided in software on the DSNIC.

The message latency of the DSNIC is dictated by the efficiency of the software, which currently relies on standard OS facilities. Major performance improvement can be obtained by avoiding the use of OS facilities that require context switches, i.e., kernel calls, task switches, and interrupts; and obtaining adequate alternatives for them. An example implementation which avoids these standard facilities is presented in [20]. This work uses polling to avoid interrupts and provides an alternative concept of processes, which allows efficient context switching. Efficient context switching improves latency and reduces CPU loading. CPU loading could be improved further by offloading functionality to the communication board.

Acknowledgements

We are very grateful for the support of the European Union through the Esprit ARCHES project.

References

[1] *The ESPRIT ARCHES Project, The Application, Refinement and Consolidation of HIC Exploiting Standards.* ESPRIT P 20693

[2] *The ATLAS Technical Proposal.* CERN/LHCC/94-43, LHCC/P2, December 1994. ISBN 92-9083-067-0

[3] *DSNIC : The DS Network Interface Card.* `http://www.cern.ch/HSI/dshs/dsnic/dsnic.html`.

[4] Eicken, T. von. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. University of California at Berkeley, PhD, 1993.

[5] Haas, S., D.A. Thornley, M. Zhu, R.W. Dobinson, R. Heeley, N.A.H. Madsen, B. Martin. Results from the Macramé 1024 Node Switching Network. *Computer Physics Communications*, April 1997. CHEP '97, Berlin, Germany.

[6] Haas, S., D.A. Thornley, M. Zhu, R.W. Dobinson, R. Heeley, B. Martin. *Results from the Macramé 1024 Node IEEE 1355 Switching Network*. European Multimedia, Microprocessor and Electronics Conference, November 1997. EMMSEC97, Florence, Italy.

[7] Heeley, R. *Real Time HEP Applications using T9000 Transputers, Links and Switches*. University of Liverpool, PhD, October 1996.

[8] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).

[9] Hockney, R., M. Berry. *Public International Benchmarks for Parallel Computers*. PARKBENCH Committee: Report-1, February 1994. `http://www.netlib.org/parkbench/`.

[10] *IEEE Std 1355-1995, IEEE Standard for Heterogeneous InterConnect (HIC) (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)*. IEEE Computer Society, 1995.

[11] *INFN, Instituto Nazionale Fisica Nucleare*. Sezione di Roma and University of Roma, La Sapienza, Rome, Italy. `http://www.roma1.infn.it/`.

[12] *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, Version 1.1, June 1995.

[13] *The Study of Noise on DS links*. OMI/Macramé, Esprit project 8603, Working paper 43, October 1996.

[14] Pakin, S., V. Karamcheti, A.A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MMPs. *IEEE Concurrency 1063-6552/97*, 5(2):60–73, 1997.

[15] Rumbaugh, J., et. al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[16] SGS THOMSON Microelectronics. *STC101 Parallel DS-Link Adaptor*. Engineering Data, August 1995.

[17] SGS THOMSON Microelectronics. *STC104 Asynchronous Packet Switch*. Engineering Data, April 1995.

[18] *The Transputer Data Book*, 2nd edition edition, 1989.

[19] Tanenbaum, A.S. *Computer Networks*. Prentice-Hall, Inc., third and international edition edition, 1996. ISBN 0-13-394248-1.

[20] Welch, P.H., M.D. Poole. *High bandwidth occam Demonstrator for Multiprocessor Alpha/DS-link systems*. Computing Laboratory, University of Kent at Canterbury, CT2 7NF, January 1998.

[21] Zhu, M., D.A. Thornley, J. Pech, B. Martin, N.H. Madsen, R. Heeley, S. Haas, R.W. Dobinson, C.R. Anderson. *Realisation and Performance of IEEE 1355 DS and HS Link based, High Speed, Low Latency Packet Switching Networks*. RT97, September 1997. Beaune, France.